# OS-9 SYSTEM PROGRAMMER'S MANUAL
VERSION 1.1

Authors: Larry A. French
Kenneth Kaplan

Contributing Forces: Bill Phelps
Bob Doggett
Larry Crane
Doug Nicholson
Toni Salzer



Revision: E

Publication Date: April 27, 1981

TABLE OF CONTENTS


INTRODUCTION TO OS-9 LEVEL I - - - - - - - - - - - - - - 1
History and Design Philosophy - - - - - - - - - - 1
System Hardware Requirements - - - - - - - - - - - 3

THE HEART OF OS-9 - - - - - - - - - - - - - - - - - - - - - 4

MULTIPROGRAMMING - - - - - - - - - - - - - - - - - - - 6
Process Creation - - - - - - - - - - - - - - - - - 6
Process States - - - - - - - - - - - - - - - - - - 7
Execution Scheduling - - - - - - - - - - - - - - - 8
Signals - - - - - - - - - - - - - - - - - - - - - 9

MEMORY MANAGEMENT - - - - - - - - - - - - - - - - - - - 10
Memory Allocation - - - - - - - - - - - - - - - - 10
Typical Memory Map - - - - - - - - - - - - - - - - 11
Memory Modules - - - - - - - - - - - - - - - - - - 12
Memory Module Structure - - - - - - - - - - - - - - 13
Module Header Definitions - - - - - - - - - - - - - 14
Typed Module Headers - - - - - - - - - - - - - - - 16
Executable Memory Module Format Diagram - - - - - - - 17
ROMed Memory Modules - - - - - - - - - - - - - - - 18

OS-9 SYSTEM COMPONENTS - - - - - - - - - - - - - - - - 19
The Kernel - - - - - - - - - - - - - - - - - - - - 19
The Configuration Module - - - - - - - - - - - - - 20
The System Bootstrap Module - - - - - - - - - - - - 21
Other Important Modules - - - - - - - - - - - - - - 22
Running OS-9 As a ROM Based System - - - - - - - - 22

THE OS-9 I/O SYSTEM - - - - - - - - - - - - - - - - - - 24
Introduction to the I/O System - - - - - - - - - - 24
Input / Output Manager - - - - - - - - - - - - - - 25
File Managers - - - - - - - - - - - - - - - - - - - 25
Device Driver Modules - - - - - - - - - - - - - - - 26
Device Descriptor Modules - - - - - - - - - - - - - 27
Device Descriptor Module Format Diagram - - - - - - 28
Path Descriptors - - - - - - - - - - - - - - - - - 32

Random Block File Manager (RBFMAN) - - - - - - - - 33
Logical and Physical Disk Organization - - - - - - 34
Identification Sector - - - - - - - - - - - - - - 34
Disk Allocation Map Sector - - - - - - - - - - - 34
File Descriptor Sectors - - - - - - - - - - - - - 35
Directories - - - - - - - - - - - - - - - - - - - 36
RBFMAN Path Descriptor Definitions - - - - - - - - 36

Sequential Character File Manager - - - - - - - - - 38
SCFMAN Path Descriptor Definitions - - - - - - - - 38
SCFMAN Line Editing Features - - - - - - - - - - - 40

INTERRUPT PROCESSING - - - - - - - - - - - - - - - - 42
IRQ Automatic Polling System - - - - - - - - - - - - 44
Writing Interrupt-Driven Device Drivers - - - - - - - 46

Writing Assembly Language Programs - - - - - - - - - 47
Using Position-Independent Code - - - - - - - - - - 47
Standard Input/Output Paths - - - - - - - - - - - 47
How To Select Addressing Modes - - - - - - - - - - 48
Machine Stack Requirements - - - - - - - - - - - - 48
Interrupt Masks - - - - - - - - - - - - - - - - - 48
Example Program - - - - - - - - - - - - - - - - - 49

OS-9 Service Request Descriptions - - - - - - - - - - 50

User Mode Function Request Descriptions - - - - - - 52
F$ABIT Set bits in an allocation bit map - - - 52
F$CHAN Chain to a new program - - - - - - - - - 53
F$CNAM Compare two names - - - - - - - - - - - 55
F$CRC Compute CRC value - - - - - - - - - - - 56
F$DBIT Deallocate in a bit map - - - - - - - - 57
F$EXIT Terminate calling process - - - - - - - 58
F$FORK Create a new process - - - - - - - - - 59
F$ICPT Set up a signal intercept trap - - - - - 61
F$ID Get process / user ID - - - - - - - - - - 62
F$LINK Link to memory module - - - - - - - - - 63
F$LOAD Load module(s) from a file - - - - - - - 64
F$MEM Resize data memory area - - - - - - - - 65
F$PERR Print error message - - - - - - - - - - 66
F$PNAM Parse a path name - - - - - - - - - - - 67
F$SBIT Search an allocation bit map - - - - - - 68
F$SEND Send a signal to another process - - - - 69
F$SLEP Put calling process to sleep - - - - - - 70
F$SPRI Set process priority - - - - - - - - - - 71
F$SSVC Install a function request - - - - - - - 72
F$SSWI Set SWI vector - - - - - - - - - - - - 74
F$STIM Set system date and time - - - - - - - - 75
F$TIME Get system date and time - - - - - - - - 76
F$UNLK Unlink a module - - - - - - - - - - - - 77
F$WAIT Wait for child to die - - - - - - - - - - 78

System Mode Function Request Descriptions - - - - - 79
F$A64 Allocate Process/Path Descriptor - - - - 79
F$APRC Enter Active Process Queue - - - - - - - 80
F$F64 Return Process/Path Descriptor - - - - - - 81
F$IODL Delete I/O Module - - - - - - - - - - - - 82
F$IOQU Enter I/O Queue - - - - - - - - - - - - - 83
F$IRQ Add Device to the IRQ polling table - - - 84
F$NPRC Start Next Process - - - - - - - - - - - 85
F$R64 Return Process/Path Descriptor - - - - - - 86
F$SRQM System Memory Request - - - - - - - - - - 87
F$SRTM System Return Memory - - - - - - - - - - 88
F$VMOD Validate Module - - - - - - - - - - - - - 89

TABLE OF CONTENTS - CONTINUED


Input/Output Request Descriptions - - - - - - - - - - - - 90
- I$ATCH Attach a new device to the system - - - - - 90
- I$CDIR Change working directory - - - - - - - - - - - 91
- I$CLOS Close a path to a file / device - - - - - - - 92
- I$CREA Create a path to a new file/device - - - - 93
- I$DLET Delete a file - - - - - - - - - - - - - - - - - - 95
- I$DTCH Detach a device from the system - - - - - - 96
- I$DUP Duplicate an I/O path - - - - - - - - - - - - - 97
- I$GSTT Get file or device status - - - - - - - - - - 98
- I$MDIR Make a new directory - - - - - - - - - - - - - 102
- I$OPEN Open an I/O path to a file/device - - - - - 103
- I$READ Read data from a file/device - - - - - - - - 104
- I$RDLN Read a text line from a file/device - - - - 105
- I$SEEK Reposition logical file pointer - - - - - - 106
- I$SSTT Set file/device status - - - - - - - - - - - - 107
- I$WRIT Write data to a file/device - - - - - - - - - 110
- I$WRLN Write a line of text to a file/device - - 111

APPENDIX A: Writing RBF-type Device Drivers - - - - A1
- Static Storage Definitions - - - - - - A1
- Device Driver Subroutines - - - - - - - A4

APPENDIX B: Writing SCF-type Device Drivers - - - - B1
- Static Storage Definitions - - - - - - B1
- Device Driver Subroutines - - - - - - - B3

APPENDIX C: Writing A System Bootstrap Module - - - C1

APPENDIX D: Memory Module Format Diagrams - - - - - - D1
- Executable Module - - - - - - - - - - - - - D1
- Device Descriptor Module - - - - - - - D2
- Configuration Module - - - - - - - - - - D3

APPENDIX E: OS-9 Floppy Disk Formats - - - - - - - - - - E1

APPENDIX F: Sample Assembly Language Programs - - - F1
- CLOCK Module - - - - - - - - - - - - - - - - - F1
- SYSGO Module - - - - - - - - - - - - - - - - - F5
- TERM Module - - - - - - - - - - - - - - - - - F6
- ACIA Module - - - - - - - - - - - - - - - - - F7
- PRINTERR Module - - - - - - - - - - - - - - - F14

APPENDIX G: Service Request Summary (code order) - - G1

APPENDIX H: OS-9 Error Codes - - - - - - - - - - - - - - - H1

# INTRODUCTION TO OS-9

OS-9 Level One is a versatile multiprogramming / multitasking operating system for computers utilizing the Motorola 6809 microprocessor. It is well-suited for a wide range of applications on 6809 computers of almost any size or complexity. Its main features are:

* Comprehensive management of all system resources: memory, input/output and CPU time.

* A powerful user interface that is easy to learn and use.

* True multiprogramming operation.

* Efficient operation in typical microcomputer configurations.

* Expandable, device-independent unified I/O system.

* Full support for modular ROMed software.

* Upward and downward compatability.

This manual is intended to provide the information necessary to install, maintain, expand, or write assembly-language software for OS-9 systems. It assumes that the reader is familiar with the 6809 architecture, instruction set, and assembly language.

## HISTORY AND DESIGN PHILOSOPHY

OS-9 is one of the results of the BASIC09 Advanced 6809 Programming Language development effort undertaken by Microware and Motorola from 1978 to 1980. During the course of the project it became evident that a fairly sophisticated operating system would be required to support BASIC09 and similar high-performance 6809 software.

OS-9's design was loosely modeled after Bell Telephone Laboratories' "UNIX" operating system, which is becoming widely recognized as a standard for mini and micro multiprogramming operating systems because of its versatility and relatively simple, yet elegant structure. Even though a "clone" of UNIX for the 6809 is relatively easy to implement, there are a number of problems with this approach. UNIX was designed for fairly large-scale minicomputers (such as large PDP-11s) that have high CPU throughput, large fast disk storage devices and a static I/O environment. Also, UNIX is not particulary time or disk-storage efficient, especially when used with low-cost disk drives.

For these reasons, OS-9 was designed to retain the overall concept and user interface of UNIX, but its implementation is considerably different. OS-9's design is tailored to typical microcomputer performance ranges and operational environments. As an example, OS-9, unlike UNIX, does not dynamically swap running programs on and off disk . This is because floppy disks and many lower-cost Winchester-type hard disks are simply too slow to do this efficiently. Instead, OS-9 always keeps running programs in memory and emphasizes more efficient use of available ROM or RAM.

OS-9 also introduces some important new features that are intended to make the most of the capabilities of third-generation microprocessors, such as support of reentrant, position-independant software that can be shared by several users simultaneously to reduce overall memory requirements.

Perhaps the most innovative part of OS-9 is its "memory module" management system, which provides extensive support for modular software, particularly ROMed software. This will play an increasingly important role in the future as a method of reducing software costs. The "memory module" and LINK capabilities of OS-9 permit modules to be automatically identified, linked together, shared, updated or repaired. Individual modules in ROM which are defective may be repaired (without reprogramming the ROM) by placing a "fixed" module with the same name, but a higher revision number into memory. Memory modules have many other advantages, for example, OS-9 can allow several programming languages to share a common math subroutine module (such as Motorola's new MC6839 floating point subroutine ROM). This same module could automatically be replaced with a module containing drivers for a hardware arithmetic processor without any change to the programs which call the module.

Users experienced with UNIX should have little difficulty adapting to OS-9. Here are some of the main differences between the two systems:

1. OS-9 is written in 6809 assembly language, not C. This improves program size and speed characteristics.

2. OS-9 was designed for a mixed RAM/ROM microcomputer memory environment and more effectively supports reentrant, position-independent code.

3. OS-9 introduces the "memory module" concept for organizing object code with built-in dynamic inter-module linkage.

4. OS-9 supports multiple file managers: modules that interface a class of devices to the file system.

5. "Fork" and "Execute" calls are faster and more memory efficient than the UNIX equivalents.

## SYSTEM HARDWARE REQUIREMENTS

The OS-9 operating system is made up of "building blocks" called modules which are automatically located and linked together when the system starts up. This makes it extremely easy to reconfigure the system. For example, reconfiguring the system to handle additional devices is simply a matter of placing the corresponding modules into memory. Because OS-9 is so flexible, the "minimum" hardware requirements are difficult to define. A bare-bones LEVEL I system requires 4K of ROM and 2K of RAM, which may be expanded to 56K RAM. A large LEVEL II system might have several hard disks, multiple terminals, and a megabyte of RAM using memory management hardware.

Below are the requirements for "typical" OS-9 computer systems. Actual hardware requirements may vary depending upon the particular application.

* 6809 MPU

* 4K Bytes RAM Memory for Single Board Computer Systems
  24K Bytes RAM Memory for Assembly Language Software
  40K Bytes RAM Memory for High Level Languages such as BASIC09
  (RAM Must Be Contiguous From Zero Up)

* 4K Bytes of ROM: 2K must be addressed at $F800 - $FFFF, the other 2K is position-independant and self-locating.

  Some disk systems may require three 2K ROMs.

  Single board computers and other ROM based systems may require an additional 6K of ROM for modules which are normally loaded from the bootstrap file (6K includes SHELL and DEBUG modules).

* Disk or cassette tape I/O device.

* Console terminal and interface using serial, parallel, or memory mapped video.

* Optional printer using serial or parallel interface.

* Optional real-time clock hardware.

I/O device controller addresses can be located anywhere in the memory space, however it is good practice to place them as high as possible to maximize RAM expansion capability. Standard Microware-supplied OS-9 packages for computers made by popular manufacturers usually conform to the system's customary memory map.

## THE KERNEL AND ITS BASIC FUNCTIONS

The heart of OS-9 is called the "kernel" which serves as the system's administrator, supervisor, and resource manager. It is about 3K bytes long and normally resides in ROM with 2K at the highest memory addresses ($F800 - $FFFF). Its main fuctions are:

1. System initialization after restart.

2. Service request processing.

3. Memory management.

4. MPU management (multiprogramming).

5. Interrupt processing.

Notice that input/output functions were not included in the list above; this is because the kernel does not directly process them. Instead, there is a separate I/O system constructed from a number of standard (or user-supplied) program modules selected to match the computer's specific hardware configuration. This is why OS-9 can be easily "tailored" to almost any 6809 computer's hardware configuration. The kernel passes I/O service requests directly to another subsystem called the "Input/Output Manager", or "IOMAN". Its function is to decode the I/O service request to select a specific "file manager" and a specific "I/O driver", which do the actual processing. The file managers and I/O drivers are also modules selected for the specific system configuration.

After a hardware reset, the kernel will initialize the system which involves such things as locating ROMs in memory, determining the amount of RAM available, loading any required modules not already in ROM from the bootstrap device, and other related tasks.

Service requests (system calls) are used to communicate between OS-9 and assembly-language-level programs for such things as allocating memory, creating new processes, etc. In addition to these callable functions, there are other "real-time" functions such as time-slicing, timekeeping, and interrupt service, which are automatic and occur routinely during normal system operation.

Memory management is a very important operating system responsibility. One way in which OS-9 is different than other operating systems is that it manages both the physical assignment AND the logical contents of memory, by using entities called "memory modules". All programs are loaded in memory module format, allowing OS-9 to maintain a directory which contains the name, address, and other related information about each module in

memory. These structures are the foundation of OS-9's modular software environment. Some of its advantages are: automatic run-time "linking" of programs to libraries of utility modules; automatic "sharing" of reentrant programs; replacement of small sections of large programs for update or correction (even when in ROM); etc.

OS-9 is a multiprogramming operating system, which means that several independent programs called "processes" can be executed simultaneously. Each process can have access to any system resource by issuing appropriate service requests to OS-9. Multiprogramming uses a hardware real-time clock that generates interrupts at a regular rate of about 10 times per second. MPU time is therefore divided into periods typically 100 milliseconds in duration. This basic time unit is called a "tick". Processes that are "active" (meaning not waiting for some event) are run for a specific system-assigned period called a "time slice". The duration of the time slice depends on a process's priority value relative to the priority of all other active processes. Many OS-9 service requests are available to create, terminate, and control processes.

Interrupt processing is another important function of the kernel. All hardware interrupts are vectored to specific processing routines. IRQ interrupts are handled by a prioritized polling system which automatically identifies the source of the interrupt and dispatches to the associated user or system defined service routine. The real-time clock will generate IRQ interrupts. SWI, SWI2, and SWI3 interrupts are vectored to user-definable addresses which are "local" to each procedure, except that SWI2 is normally used for OS-9 service requests calls. The NMI and FIRQ interrupts are not normally used and are vectored through a RAM address to an RTI instruction.

At any given moment, OS-9 is in one of two states: "user state" when a user process is in execution, and "system state" during execution of OS-9 routines which occurs after any system service request or a hardware interrupt.

# MULTIPROGRAMMING

This section of the manual deals with the following aspects of multiprogramming under the OS-9 operating system: process creation, process states, execution scheduling, and signals as a means of inter-process communication.

## PROCESS CREATION

New processes are created when an existing process executes a "fork" service request. Its main argument is the name of the program module (called the "primary module") that the new process is to initially execute. OS-9 first attempts to find the module in the "module directory", which includes the names of all program modules already present in memory. If the module cannot be found there, OS-9 usually attempts to load into memory a mass-storage file using the requested module name as a file name.

Once the module has been located, a data structure called a "process descriptor" is assigned to the new process. The process descriptor is a 64-byte package that contains information about the process, its state, memory allocations, priority, queue pointers, etc. The process descriptor is automatically initialized and maintained by OS-9. The process itself has no need, and is not permitted to access the descriptor.

The next step in the creation of a new process is allocation of data storage (RAM) memory for the process. The primary module's header contains a storage size value that is used unless the "fork" system call requested an optionally larger size. OS-9 then attempts to allocate a CONTIGUOUS memory area of this size from the free memory space.

If any of the previous steps cannot be performed, creation of the new process is aborted, and the process that originated the "fork" is informed of the error. Otherwise, the new process is added to the active process queue for execution scheduling.

The new process is also assigned a unique number called a "process ID" which is used as its identifier. Other processes can communciate with it by referring to its ID in various system calls. The process also has associated with it a "user ID" which is used to identify all processes and files belonging to a particular user. The user ID is inherited from the parent process.

Processes terminate when they execute an "EXIT" system service request, or when they receive fatal signals. The process termination closes any open paths, deallocates its memory, and unlinks its primary module.

## PROCESS STATES

Each process can be in one of three states:

- ACTIVE - The process is active and ready for execution.
- WAITING - The process is suspended until some event occurs.
- SLEEPING - The process is suspended for a specific period of time.

There is a queue for each process state. The queue is a linked list of the "process descriptors" of processes in the corresponding state. State changes are performed by moving a process descriptor to another queue.

### The Active State:

This state includes all "runnable" processes, which are given time slices for execution according to their relative priority with respect to all other active processes. The scheduler uses a pseudo-round-robin scheme that gives all active processes some CPU time, even if they have a very low relative priority.

### The Wait State:

This state is entered when a process executes a WAIT system service request. The process remains suspended until the death of any of its descendant processes, or, until it receives a signal.

### The Sleeping State

This state is entered when a process executes a SLEEP service request, which specifies a time interval for which the process is to remain suspended. The process remains asleep until the specified time has elapsed, or until a signal is received.

## EXECUTION SCHEDULING

The kernel contains a scheduler that is responsible for allocation of CPU time to active processes. OS-9 uses a scheduling algorithm that ensures all processes get some execution time.

All active processes are members of the "active process queue", which is kept sorted by process "age". Age is a count of how many process switches have occurred since the process' last time slice. When a process is moved to the active process queue from another queue, its "age" is initialized by setting it to the process' assigned priority, i.e., processes having relatively higher priority are placed in the queue with an artificially higher age. Also, whenever a new process is activated, the ages of all other processes are incremented.

Upon conclusion of the currently executing process' time slice, the scheduler selects the process having the highest age to be executed next. Because the queue is kept sorted by age, this process will be at the head of the queue. At this time the ages of all other active processes are incremented (ages are never incremented beyond 255).

An exception is newly-active processes that were previously deactivated while they were in the system state. These processes are noted and given higher priority than others because they are usually executing critical routines that affect shared system resources and therefore could be blocking other unrelated processes.

When there are no active processes, the kernel will set itself up to handle the next interrupt and then execute a CWAI instruction, which decreases interrupt latency time.

## SIGNALS

"Signals" are an asynchronous control mechanism used for inter-process communication and control. A signal behaves like a software interrupt in that it can cause a process to suspend a program, execute a specific routine, and afterward return to the interrupted program. Signals can be sent from one process to another process (by means of the SEND service request), or they can be sent from OS-9 system routines to a process.

Status information can be conveyed by the signal in the form of a one-byte numeric value. Some of the signal "codes" (values) have predefined meanings, but all the rest are user-defined. The defined signal codes are:

0 = KILL (non-interceptable process abort)
1 = WAKEUP - wake up sleeping process
2 = KEYBOARD ABORT
3 = KEYBOARD INTERRUPT
4 - 255 USER DEFINED

When a signal is sent to a process, the signal is noted and saved in the process descriptor. If the process is in the sleeping or waiting state, it is changed to the active state. It then becomes eligible for execution according to the usual MPU scheduler criteria. When it gets its next time slice, the signal is processed.

What happens next depends on whether or not the process had previously set up a "signal trap" (signal service routine) by executing an INTERCEPT service request. If it had not, the process is immediately aborted. It is also aborted if the signal code is zero. The abort will be deferred if the process is in system mode: the process dies upon its return to user state.

If a signal intercept trap has been set up, the process resumes execution at the address given in the INTERCEPT service request. The signal code is passed to this routine, which should terminate with an RTI instruction to resume normal execution of the process.

NOTE: "Wakeup" signals activate a sleeping process: they DO NOT vector through the intercept routine.

If a process has a signal pending (usually because it has not been assigned a time slice since the signal was received), and some other process attempts to send it another signal, the new signal is aborted and the "send" service request will return an error status. The sender should then execute a "sleep" service request for a few ticks before attempting to resend the signal, so the destination process has an opportunity to process the previously pending signal.

# MEMORY MANAGEMENT

## MEMORY ALLOCATION

All usable RAM memory must be contiguous from address Ø upward. During the OS-9 start-up sequence the upper bound of RAM is detemined by an automatic search, or from the configuration module. Some RAM is reserved by OS-9 for its own data structures at the top and bottom of memory. The exact amount depends on the sizes of system tables that are specified in the configuration module.

All other RAM memory is pooled into a "free memory" space. Memory space is dynamically taken from and returned to this pool as it is allocated or deallocated for various purposes. The basic unit of memory allocation is the 256-byte "page". Memory is always allocated in whole numbers of pages.

The data structure used to keep track of memory allocation is a 32-byte bit-map located at addresses $Ø1ØØ - $Ø11F. Each bit in this table is associated with a specific page of memory. Bits are cleared to indicate that the page is free and available for assignment, or set to indicate that the page is in use or that no RAM memory is present at that address.

Automatic memory allocation occurs when:

1. Program modules are loaded into RAM.
2. Processes are created.
3. Processes request additional RAM.
4. OS-9 needs I/O buffers, larger tables, etc.

All of the above usually have inverse functions that cause previously allocated memory to be deallocated and returned to the free memory pool.

In general, memory is allocated for program modules and buffers from high addresses downward, and for process data areas from lower addresses upward.

TYPICAL MEMORY MAP

```
+------------------------+  <- $FFFF
|                        |
|     OS-9 ROMS (4K)     |
|                        |
+------------------------+  <- $F000
|                        |
|  I/O DEVICE ADDRESSES  |
|                        |
+------------------------+  <- $E000
|                        |
|     SPACE FOR MORE     |
|     OPTIONAL ROMS      |
|                        |
+------------------------+  <- END OF RAM MEMORY
|                        |
|     FILE MANAGERS,     |
|  DEVICE DRIVERS, ETC.  |
|   (APPROXIMATELY 6K)   |
|                        |
+------------------------+
|                        |
|       SHELL (1K)       |
|                        |
+------------------------+
|                        |
|  OS-9 DATA STRUCTURES  |
|  (APPROXIMATELY 1K)    |
|                        |
+------------------------+
|                        |
|    FREE MEMORY FOR     |
|     GENERAL USE        |
|                        |
+------------------------+ <- $0400
|                        |
|  OS-9 DATA STRUCTURES  |
|   AND DIRECT PAGE      |
|                        |
+------------------------+ <- $0000 BEGINNING OF RAM MEMORY
```

The map above is for a "typical" system. Actual memory sizes and addresses may vary depending on the exact system configuration.

## MEMORY MODULES

All programs used on OS-9 systems must use the memory module format and conventions. Many of OS-9's extraordinary capabilities would not be possible without them.

The memory module concept allows OS-9 to manage the logical contents as well as the physical contents of memory. The basic idea is that all programs are discrete, named units. The operating system keeps track of modules which are in memory at all times by use of a "module directory". It contains the addresses and a count of how many processes are using each module. When modules are loaded into memory, they are added to the directory. When they are no longer needed, their memory is deallocated and their name removed from the directory (except ROMs, which are discussed later). In many respects, modules and memory in general, are managed just like a disk. In fact, the disk and memory management sections of OS-9 share many subroutines.

Each module has three parts; a module header, module body and a cyclic-redundancy-check (CRC) value. The header contains information that describes the module and its use. This information includes: the modules size, its type (machine language, BASIC09 compiled code, etc); attributes (executable, reentrant, etc), data storage memory requirements, execution starting address, etc. The CRC value is used to verify the integrity of a module.

There are several different kinds of modules, each type having a different usage and function. Modules do not have to be complete programs, or even 6809 machine language. They may contain BASIC09 "I-code", constants, single subroutines, subroutine packages, etc. The main requirements are that modules do not modify themselves and that they be position-independent so OS-9 can load or relocate them wherever memory space is available. In this respect, the module format is the OS-9 equivalent of "load records" used in older-style operating systems.

## MEMORY MODULE STRUCTURE

At the beginning (lowest address) of the module is the module header, which can have several forms depending on the module's usage. The header is described more thoroughly later in this section. Following the header is the program/constant section which is usually pure code. The module name string is included somewhere in this area. The last three bytes of the module are a three-byte Cyclic Redundancy Check (CRC) value used to verify the integrity of the module.

MODULE FORMAT

```
+---------------------+
|                     |
|    MODULE HEADER    |
|                     |
+---------------------+
|                     |
|       PROGRAM       |
|    OR CONSTANTS     |
|                     |
+---------------------+
|         CRC         |
+---------------------+
```

The 24-bit CRC is performed over the entire module from the first byte of the module header to the byte just before the CRC itself. The CRC polynomial used is $800FE3.

Because most OS-9 family software (such as the assembler) automatically generate the module header and CRC values, the programmer usually does not have to be concerned with writing routines to generate them.

## MODULE HEADER DEFINITIONS

The format of module headers may seem complicated at first, but in practice they are simple to create and use because OS-9 family software such as BASIC09, the assembler, and many utility programs automatically generate modules and headers. It is a good idea to have a general understanding of what they look like and how they work. The first nine bytes of all module headers are identical:

| MODULE OFFSET | DESCRIPTION |
|---|---|
| \$0,\$1 = | Sync Bytes (\$87,\$CD). These two constant bytes are used to locate modules. |
| \$2,\$3 = | Module Size. The overall size of the module in bytes (includes CRC). |
| \$4,\$5 = | Offset to Module Name. The address of the module name string relative to the start (first sync byte) of the module. The name string can be located anywhere in the module and consists of a string of ASCII characters having the sign bit set on the last character. |
| \$6 = | Module Type/Langauge Type. See text. |
| \$7 = | Attributes/Revision Level. See text. |
| \$8 = | Header Check. The one´s compliment of the vertical parity (exclusive OR) of the previous eight bytes. |

### Type/Language Byte

The module type is coded into the four most significant bits of byte 6 of the module header. Eight types are pre-defined by convention, some of which are for OS-9´s internal use only. The type codes are:

| | |
|---|---|
| \$1 | Program module |
| \$2 | Subroutine module |
| \$3 | Multi-module |
| \$4 | Data module |
| \$5-\$B | User-definable |
| \$C | OS-9 System module |
| \$D | OS-9 File Manager module |
| \$E | OS-9 Device Driver module |
| \$F | OS-9 Device Descriptor module |

NOTE: 0 is not a legal type.

The four least significant bits of byte 6 describe the language type as listed below:

| | |
|---|---|
| 0 | DATA |
| 1 | 6809 object code |
| 2 | BASIC09 I-code |
| 3 | PASCAL P-code |
| 4 | COBOL I-code |
| 5-15 | Reserved for future use |

The purpose of the language type is so high-level language run-time systems can verify that a module is of the correct type before execution is attempted. BASIC09, for example may run either I-code or 6809 machine language procedures arbitrarily by checking the language type code.

## Attribute/Revision Byte

The upper four bits of this byte are reserved for module attributes. Currently, only bit 7 is defined, and when set indicates the module is reentrant and therefore "sharable".

The lower four bits are a revision level from zero (lowest) to fifteen. If more than one module has the same name, type, language, etc., OS-9 only keeps in the module directory the module having the highest revision level. This is how ROMed modules can be replaced or patched: you load a new, equivalent module having a higher revision level. Because all modules locate each other by using the LINK system call which searches the module directory by name, it always returns the latest revision of the module, wherever it may be.

NOTE: A previously linked module can not be replaced until all processes which linked to it have unlinked it (after its link count goes to zero).

## TYPED MODULE HEADERS

As mentioned before, the first nine bytes of the module header are defined identically for all module types. There is usually more header information immediately following, the layout and meaning varies depending on the specific module type. Module types \$C - \$F are used exclusively by OS-9. Their format is given elsewhere in this manual.

The module type illustrated below is the general-purpose "user" format that is commonly used for OS-9 programs that are called using the FORK or CHAIN system calls. These modules are the "user-defined" types having type codes of Ø through 9. They have six more bytes in their headers defined as follows:

MODULE OFFSET — DESCRIPTION

\$9,\$A = Execution Offset. The program or subroutine's starting address, relative to the first byte of the sync code. Modules having multiple entry points (cold start, warm start, etc.) may have a branch table starting at this address.

\$B,\$C = Permanent Storage Requirement. This is the minimum number of bytes of data storage required to run. This is the number used by FORK and CHAIN to allocate a process' data area.

If the module will not be directly executed by a CHAIN or FORK service request (for instance a subroutine package), this entry is not used by OS-9. It is commonly used to specify the maximum stack size required by reentrant subroutine modules. The calling program can check this value to determine if the subroutine has enough stack space.

## EXECUTABLE MEMORY MODULE FORMAT

```
MODULE
OFFSET
          +-------------------------------+    ---+----------+---
  $00     !                               !       !          !
          +--     Sync Bytes ($87CD)    --+       !          !
  $01     !                               !       !          !
          +-------------------------------+       !          !
  $02     !                               !       !          !
          +--     Module Size (bytes)   --+       !          !
  $03     !                               !       !          !
          +-------------------------------+       !          !
  $04     !                               !       !          !
          +--     Module Name Offset    --+    header        !
  $05     !                               !    parity        !
          +-------------------------------+       !          !
  $06     !     Type       !   Language   !       !          !
          +-------------------------------+       !          !
  $07     !  Attributes    !   Revision   !       !          !
          +-------------------------------+    ---+---    module
  $08     !      Header Parity Check      !                 CRC
          +-------------------------------+                  !
  $09     !                               !                  !
          +--      Execution Offset     --+                  !
  $0A     !                               !                  !
          +-------------------------------+                  !
  $0B     !                               !                  !
          +--  Permanent Storage Size   --+                  !
  $0C     !                               !                  !
          +-------------------------------+                  !
  $0D     !  (Add'l optional header       !                  !
          !   extensions located here)    !                  !
          !                               !                  !
          !   .  .  .  .  .  .  .  .  .   !                  !
          !                               !                  !
          !                               !                  !
          !        Module Body            !                  !
          ! object code, constants, etc.  !                  !
          !                               !                  !
          !                               !                  !
          +-------------------------------+                  !
          !                               !                  !
          +--                           --+                  !
          !       CRC Check Value         !                  !
          +--                           --+                  !
          !                               !                  !
          +-------------------------------+    ----------+---
```

## ROMED MEMORY MODULES

When OS-9 starts after a system reset, it searches the entire memory space for ROMed modules. It detects them by looking for the module header sync code ($87,$CD) which are unused 6809 opcodes. When this byte pattern is detected, the header check is performed to verify a correct header. If this test succeeds, the module size is obtained from the header and a 24-bit CRC is performed over the entire module. If the CRC matches correctly, the module is considered valid and it is entered into the module directory. The chances of detecting a "false module" are virtually nil.

In this manner all ROMed modules present in the system at startup are automatically included in the system module directory. Some of the modules found initially are various parts of OS-9: file managers, device driver, the configuration module, etc.

After the module search OS-9 links to whichever of its component modules that it found. This is the secret of OS-9's extraordinary adaptablity to almost any 6809 computer; it automatically locates its required and optional component modules, wherever they are, and rebuilds the system each time that it is started.

ROMs containing non-system modules are also searched so any user-supplied software is located during the start-up process and entered into the module directory.

## OS-9 SYSTEM COMPONENTS: THE MAJOR MODULES

OS-9 is composed of a number of modules. Some of the component modules are required. Others are optional and included in a particular computer to meet a desired performance level or to support specific hardware or I/O devices. Below is a list of the required and optional modules:

| | |
|---|---|
| REQUIRED: | Kernel (2K must reside at $F800-$FFFF)<br>Configuration Module |
| OPTIONAL: | System Bootstrap Module<br>Input Output Manager (IOMAN)<br>File Manager (SCFMAN or RBFMAN)<br>Device Drivers<br>Device Descriptors<br>System Initialization Module (SYSGO) |

## THE KERNEL

The kernel is the heart of OS-9. Its purpose is to:

1. Initialize the system after reset
2. Process system calls.
3. Manage RAM memory.
4. Manage the module directory.
5. Service interrupts.

The kernel's total size is approximately 3K bytes, which is partitioned into two sections. The first section called "OS9" contains the interrupt vector tables which must be located at $FFE0 through $FFFF. The second section called "OS9P2" can be located anywhere in memory. Both sections must be in ROM! If the target system bootstraps other OS-9 component modules into RAM from disk or tape (standard Microware supplied versions do so), OS-9 fits into a 4K ROM or a pair of 2K ROMS, leaving an extra 1K for a bootstrap device driver module. Some lengthy disk driver modules that are larger than 1K require an additional ROM.

## THE CONFIGURATION MODULE

This module defines system startup parameters and must also be in ROM (it fits in the same ROM as the kernel). It is a non-executable module named "INIT" and has type "system" (code $C). It is scanned once during the system startup. It begins with the standard header followed by:

| MODULE OFFSET | |
|---|---|
| $9,$A,$B | This location contains the forced limit of free RAM memory. It may be used to override OS-9's automatic top-of-RAM search so that memory may be reserved for special purposes. |
| $C | Number of entries to create in the IRQ polling table. One entry is required for each interrupt generating device control register. |
| $D | Number of entries to create in the system device table. One entry is required for each device in the system. |
| $E,$F | Offset to the intitial startup module name string. This is the name of the first module to be executed after startup, usually "SYSGO". There must always be a startup module. |
| $10,$11 | Offset to the initial standard path string (typically /TERM). This path is opened as the standard paths for the initial startup module. This location should contain zero if there is none. |
| $12,$13 | Offset to the default directory name string (normally /D0). This device is assumed when device names are ommited from pathlists. If the system will not initialy use a mass storage device (such as in ROM based systems) this location should contain a zero. |
| $14,$15 | Offset to bootstrap module name string. If OS-9 cannot locate any of its other required modules in ROM, it will call the bootstrap module which attempts to load them from a mass-storage device. |
| $16 to N | All name strings go here, followed by three CRC bytes. Name strings must have the sign bit (bit 7) of the last character set. |

(also see the memory module format diagrams in Appendix D)

The "default" configuration module included in the OS-9 kernel ROM can be replaced by a user-supplied module. Just make your own module according to the specifications above, giving it a revision level (byte 7 of the header) of one or higher. This module must be in ROM.

## THE SYSTEM BOOTSTRAP MODULE

This module loads several of OS-9's required and optional modules from the bootstrap file on a mass-storage device. During the kernel's startup sequence, it tries to link to other modules such as IOMAN. If they are not in memory, the kernel will call the bootstrap module to load them from mass-storage. Some OS-9 systems will have all required modules in ROM so the bootstrap module is not needed. Also see "Running OS-9 As a ROM Based System".

In many systems, it is not cost efffective or otherwise practical to place all of OS-9 into ROM. In these systems, only the kernel, configuration, and bootstrap modules are ROMed. When the system is reset, the kernel will call the bootstrap module which attempts to load the boostrap file into memory from a mass-storage device such as disk or tape. The bootstrap file contains all modules required by OS-9 that are not already in ROM, and other optional modules which should be loaded into memory at startup time. Below is a list of the modules which are typically loaded from the bootstrap file (the actual contents may vary for a particular system or hardware configuration):

| MODULE NAME | |
|---|---|
| IOMAN | Input Output Manager |
| RBF | Random Block File Manager |
| SCF | Sequential Character File Manager |
| DSK | Disk Driver (name varies with controller) |
| ACIA | ACIA Driver |
| PIA | PIA Driver |
| Clock | Real-time-clock interface module |
| DØ | Drive zero device descriptor (DISK) |
| D1 | Drive one device descriptor (DISK) |
| TERM | Console terminal device descriptor (ACIA) |
| T1 | Secondary terminal device descriptor (ACIA) |
| P | Printer device descriptor (PIA) |
| P1 | Secondary printer device descriptor (ACIA) |
| SYSGO | System startup module |
| SHELL | System command interpreter |

## OTHER IMPORTANT MODULES

It is possible for very simple computers that don't have standard I/O devices to get by with just the kernel. But most practical systems will be interfaced to a terminal and some kind of mass storage or communications device. Additional OS-9 modules interface the kernel to the I/O system. These are the I/O Manager ("IOMAN"), one or more File Manager modules, one or more Device Driver modules, and one or more Device descriptors.

## RUNNING OS-9 AS A ROM BASED SYSTEM

For many aplications using OS-9, a mass storage device such as a disk is either too costly or otherwise inappropriate. Certain applications require that the operating system be functional before a mass-storage device is used; for instance, it may be necessary to be able to run a system diagnostic program such as the interactive DEBUG module before the using the disk drives. OS-9 can eaisly be tailored to meet these needs. To run OS-9 as a ROM based system, all that is required is to adjust two parameters in the system configuration module (INIT) and put a few modules into ROM. Below is a description of how the INIT module must be adjusted:

A. If the system will not use a mass storage device, set the offset to the default mass-storage device name string to zero in the INIT module.

B. If it is not necessary to bootstrp from a mass-storage device, set the offset to the BOOT module name string to zero in the INIT module.

Below is a list of the name and approximate size of the modules which are required for OS-9 to start up and be fully functional (complete with the SHELl command interpreter):

| MODULE | SIZE | |
|---|---|---|
| OS9 | $76F | OS-9 Part One |
| OS9P2 | $4CB | OS-9 Part Two |
| IOMAN | $646 | I/O Manager |
| SCF | $3D9 | Sequential Character File Manager |
| ACIA | $183 | ACIA device driver |
| TERM | $38 | Console terminal device descriptor |
| SYSGO | $66 | System initialization module |
| INIT | $24 | System configuration module |
| SHELL | $403 | Shell command line interpreter |
| | ------ | |
| | $1CA1 = 7329 | System ROM requirements |

This set of modules is "typical" for a single board microcomputer, the actual system requirements will vary (see the section of this manual on "SYSTEM HARDWARE REQUIREMENTS"). In some applications it may be desirable to include a real-time-clock driver module, bootstrap module (so that a mass-storage device may be used after startup), or the interactive DEBUG module for system diagnostics purposes. The name and size of these modules is given below:

| MODULE | SIZE | |
|---|---|---|
| CLOCK | $C7 | Real-time-clock driver module |
| BOOT | $280 | Bootstrap module |
| DEBUG | $7BB | Interactive DEBUG module |

NOTE: The actual size of these modules may vary slightly.

## THE OS-9 INPUT/OUTPUT SYSTEM

The OS-9 input/output system is constructed from a number of standard or user supplied modules selected to match a computer's hardware configuration. These modules are organized into a hierarchical structure like the one below:

```
                               IOMAN
                                 !
                                 !
     +---------------------------+------------------+
     !                           !                  !
     !                           !                  !

   SCFMAN                      RBFMAN            SBFMAN **

     !                           !                  !
     !                           !                  !
+----+-----+                    DISK               TAPE
!          !
!          !                     !                  !
                                 !                  !
ACIA      PIA               +----+----+        +----+----+
                            !         !        !         !
 !         !                !         !        !         !
 !         !
+---+---+                   DØ        D1      TDØ       TD1
!   !   !  P
!   !   !

T1  P1  TERM
```

The hierarchical organization makes it extremely easy to reconfigure or expand the I/O system, which is typically done by loading revised or additional modules into memory (they are automatically installed in the I/O system).

At the top level in the diagram, data is device independent and treated as if it were a stream of bytes. Data flowing up from the device level is "filtered" by each module that it passes through, so that when it reaches the top, it will appear as a stream of bytes and no longer be dependent on the way it happened to come into the system. Likewise, data flowing from the top down is "filtered" so that it will conform to any device dependencies; for example, the data may be blocked, line feeds may be inserted after carriage returns, etc.

** NOTE: SBFMAN IS NOT CURRENTLY SUPPORTED BY MICROWARE

There are four levels of modules in the OS-9 I/O system as given below:

1. I/O Manager (IOMAN)
2. File Managers (SCFMAN, RBFMAN, SBFMAN)
3. Device Drivers (ACIA, PIA, DISK, etc.)
4. Device Descriptors (TERM, D0, D1, etc.)

## THE INPUT/OUTPUT MANAGER

The Input/Output Manager (IOMAN) module provides the first level of service for I/O system calls by routing data on I/O paths from processes to/from the appropriate file managers and device drivers. It maintains two important internal OS-9 data structures: the device table and the path table. This module is universal for all OS-9 Level One systems and most users should never need to alter or replace it.

## FILE MANAGERS

OS-9 systems can have one or more File Manager modules. The function of a file manager is to process the raw data stream to or from the device drivers to conform to the OS-9 standard file structure.

File managers usually buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory. They may also monitor and process the data stream, for example, adding line feed characters after carriage return characters.

The file managers are reentrant and one file manager may be used for an entire class of devices having similar operational characteristics. Standard OS-9 file managers are:

RBFMAN: The Random Block File Manager which operates random-access, block-structured devices such as disk systems, bubble memories, etc.

SCFMAN: Sequential Character File Manager which is used with single-character-oriented devices such as CRT or hardcopy terminals, printers, modems, etc.

SBFMAN: Sequential Block File Manager which drives block oriented devices that don't have random access capability, mostly tape systems.

NOTE: SBFMAN IS NOT CURRENTLY SUPPORTED BY MICROWARE

Most OS-9 systems will have two file managers: SCFMAN to handle terminals, and either RBFMAN (disk-based systems) or SBFMAN (tape-based systems). It is possible to use all three or more. Sophisticated users may wish to write their own special-purpose file managers.

## DEVICE DRIVER MODULES

The device driver modules contain a package of subroutines that perform raw I/O transfers to or from a specific type of I/O device hardware controller. These modules are usually reentrant and one copy of the module can simultaneously run several different devices which use identical I/O controllers. For example the device driver for 6850 serial interfaces is called "ACIA" and can communicate to any number of serial terminals. Provisions have been made so each "incarnation" of the driver can have its operational characteristics (such as paging, echo, backspace and delete characters, etc.) individually settable.

Device driver modules use a standard module header and are given a module type of "device driver" (code $E). The execution offset address in the module header points to a branch table that has a minimum of six (three-byte) entries. Each entry is typically a LBRA to the corresponding subroutine. The File Managers call specific routines in the device driver through this table, passing a pointer to a "path decriptor" and the hardware control register address in the MPU registers. The branch table looks like:

```
 +0 = Device Initialization Routine
 +3 = Read From Device
 +6 = Write to Device
 +9 = Get Device Status
+$C = Set Device Status
+$F = Device Termination Routine
```

For a complete description of the parameters passed to these subroutines see the file manager descriptions. Also see the appendicies on writing device drivers.

## DEVICE DESCRIPTOR MODULES

These small non-executable modules provide the system with the information it needs to use a device. They associate a specific I/O device with its:

(a) Logical name.
(b) Hardware controller address(es).
(c) Device driver name.
(d) File manager name.
(e) Initial operating parameters.

Recall that device drivers and file managers both operate on classes of devices, not specific devices. The device descriptor modules tailor their functions for a specific I/O device. One device descriptor module must exist for each logical device in the OS-9 environment.

The name of the module is the name the device is known by to the system and user (i.e. it is the device name given in pathlists). Its format consists of a standard module header that has a type "device descriptor" (code $F). The rest of the device descriptor header consists of:

$9,$A = File manager name string relative address.

$B,$C = Device driver name string relative address.

$D = Mode/Capabilities (D S PE PW PR E W R)

$E,$F,$10 = Device controller absolute physical (24-bit) address

$11 = Number of bytes ("n" bytes in intialization table)

$12,$12+n = Initialization table

The initialization table is copied into the "option section" of the path descriptor when a path to the device is opened. The values in this table may be used to define the operating parameters that are changeable by the OS9 I$GSTT and I$SSTT service requests. For example, a terminal's initialization parameters define which control characters are used for backspace, delete, etc. The maximum size of initialization table which may be used is 32 bytes. If the table is less than 32 bytes long, the remaining values in the path descriptor will be set to zero.

You may wish to add additional devices to your system. If a similar device controller already exists, all you need to do is add the new hardware and load another device descriptor. Device descriptors can be in ROM or loaded into RAM at any time.

The diagram on the next page illustrates the device descriptor module format.

```
MODULE      DEVICE DESCRIPTOR MODULE FORMAT
OFFSET

           +------------------------------+  ---+---------+---
  $0       !                              !     !         !
           +--    Sync Bytes ($87CD)    --+     !         !
  $1       !                              !     !         !
           +------------------------------+     !         !
  $2       !                              !     !         !
           +--       Module Size        --+     !         !
  $3       !                              !     !         !
           +------------------------------+     !         !
  $4       !                              !     !         !
           +--  Offset to Module Name   --+   header      !
  $5       !                              !   parity      !
           +------------------------------+     !         !
  $6       ! $F (TYPE)    !  $1 (LANG)    !     !         !
           +------------------------------+     !         !
  $7       ! Atributes    !   Revision    !     !         !
           +------------------------------+  ---+---      !
  $8       !     Header Parity Check      !               !
           +------------------------------+               !
  $9       !                              !               !
           +--  Offset to File Manager  --+               !
  $A       !        Name String           !             module
           +------------------------------+              CRC
  $B       !                              !               !
           +-- Offset to Device Driver  --+               !
  $C       !        Name String           !               !
           +------------------------------+               !
  $D       !         Mode Byte            !               !
           +------------------------------+               !
  $E       !                              !               !
           +--    Device Controller     --+               !
  $F       ! Absolute Physical Address    !               !
           +--        (24 bit)          --+               !
  $10      !                              !               !
           +------------------------------+               !
  $11      !  Initialization Table Size   !               !
           +------------------------------+               !
$12,$12+N  !                              !               !
           !   (Initialization Table)     !               !
           !                              !               !
           +------------------------------+               !
           !                              !               !
           !    (Name Strings etc)        !               !
           !                              !               !
           +------------------------------+               !
           !                              !               !
           +--                          --+               !
           !      CRC Check Value         !               !
           +--                          --+               !
           !                              !               !
           +------------------------------+  -------------+---
```

Below are the initialization table definitions for SCF and RBF type devices:

## SCF DEVICE INITIALIZATION TABLE

```
MODULE
OFFSET              ORG $12

         TABLE      EQU .   BEGINING OF OPTION TABLE
$12      IT.DVC     RMB 1   DEVICE CLASS (Ø=SCF 1=RBF 2=PIPE 3=SBF)
$13      IT.UPC     RMB 1   CASE (Ø=BOTH, 1=UPPER ONLY)
$14      IT.BSO     RMB 1   BACK SPACE (Ø=BSE, 1=BSE,SP,BSE)
$15      IT.DLO     RMB 1   DELETE (Ø=BSE OVER LINE, 1=CR)
$16      IT.EKO     RMB 1   ECHO (Ø=NO ECHO)
$17      IT.ALF     RMB 1   AUTO LINE FEED (Ø= NO AUTO LF)
$18      IT.NUL     RMB 1   END OF LINE NULL COUNT
$19      IT.PAU     RMB 1   PAUSE (Ø= NO END OF PAGE PAUSE)
$1A      IT.PAG     RMB 1   LINES PER PAGE
$1B      IT.BSP     RMB 1   BACKSPACE CHARACTER
$1C      IT.DEL     RMB 1   DELETE LINE CHARACTER
$1D      IT.EOR     RMB 1   END OF RECORD CHARACTER
$1E      IT.EOF     RMB 1   END OF FILE CHARACTER
$1F      IT.RPR     RMB 1   REPRINT LINE CHARACTER
$2Ø      IT.DUP     RMB 1   DUP LAST LINE CHARACTER
$21      IT.PSC     RMB 1   PAUSE CHARACTER
$22      IT.INT     RMB 1   INTERRUPT CHARACTER
$23      IT.QUT     RMB 1   QUIT CHARACTER
$24      IT.BSE     RMB 1   BACKSPACE ECHO CHARACTER
$25      IT.OVF     RMB 1   LINE OVERFLOW CHARACTER (BELL)
$26      IT.PAR     RMB 1   INITIALIZATION VALUE (PARITY)
$27      IT.BAU     RMB 1   BAUD RATE
$28      IT.D2P     RMB 2   ATTACHED DEVICE NAME STING OFFSET
$2A                 RMB 2   RESERVED
$2C      IT.ERR     RMB 1   INTITIAL ERROR STATUS
```

NOTES:

SCF editing functions will be "turned off" if the corresponding special character is a zero. For example, if IT.EOF was a zero, there would be no end of file character. For a full description, please see the section of this manual on "Sequential Character File Manager".

IT.PAR is typically used to intitialize the device's control register when a path is opened to it.

## RBF DEVICE INITIALIZATION TABLE

```
MODULE
OFFSET          ORG $12

$12      IT.DTP  RMB 1   DEVICE TYPE (Ø=SCF 1=RBF 2=PIPE 3=SBF)
$13      IT.DRV  RMB 1   DRIVE NUMBER
$14      IT.STP  RMB 1   STEP RATE
$15      IT.TYP  RMB 1   DEVICE TYPE (See RBFMAN path descriptor)
$16      IT.DNS  RMB 1   MEDIA DENSITY (Ø = SINGLE, 1=DOUBLE)
$17      IT.CYL  RMB 2   NUMBER OF CYLINDERS (TRACKS)
$19      IT.SID  RMB 1   NUMBER OF SURFACES (SIDES)
$1A      IT.VFY  RMB 1   Ø = VERIFY DISK WRITES
$1B      IT.SCT  RMB 2   Default Sectors/Track
$1D      IT.TØS  RMB 2   Default Sectors/Track (Track Ø)
$1F      IT.ILV  RMB 1   SECTOR INTERLEAVE FACTOR
$2Ø      IT.SAS  RMB 1   SEGMENT ALLOCATION SIZE
```

NOTES:

IT.DRV This location is used to associate a one byte integer with each drive that a controller will handle. The drives for each controller should be numbered Ø to N-1.

IT.STP This location sets the head stepping rate that will be used with a drive. The step rate should be set to the fastest value that the drive is capable of to reduce access time. Below are the values which should be used:

| STEP CODE | FD1771 | | FD1791, FD1797 | |
|---|---|---|---|---|
| | 5" | 8" | 5" | 8" |
| Ø | 4Øms | 2Øms | 3Øms | 15ms |
| 1 | 2Øms | 1Øms | 2Øms | 1Øms |
| 2 | 12ms | 6ms | 12ms | 6ms |
| 3 | 12ms | 6ms | 6ms | 3ms |

The DC-2, DC-3, GMX #58 and BFD-68A controllers use the FD1771 type chips.

The DF-2, GMX #28, GMX DMA, and DCB-4 controllers use the FD1791 or FD1797 type chips.

```
IT.TYP    Device type.

              bit Ø -- Ø = 5" floppy disk
                       1 = 8" floppy disk

              bit 6 -- Ø = Standard OS-9 format
                       1 = Non-standard format

              bit 7 -- Ø = Floppy disk
                       1 = Hard disk (may have smart controller)


IT.DNS    Density capabilities

              bit Ø -- Ø = Single bit density
                       1 = Double bit density

              bit 1 -- Ø = Single track density
                       1 = Double track density

IT.SAS    This value specifies the minimum number of  sectors  to
          be allocated at any one time.
```

For more information, please the section of this manual on RBFMAN definitions of the path descriptor.

## PATH DESCRIPTORS

Every open path is associated with a data structure called a path descriptor ("PD"). It contains the information required by the file managers and device drivers to perform I/O functions. Path descriptors are exactly 64 bytes long and are dynamically allocated and deallocated by IOMAN in response to requests from user programs.

PDs are INTERNAL data structures that are not normally referenced from user or applications programs. In fact, it is almost impossible to locate a path's PD when OS-9 is in user mode. The description of PDs is mostly of interest to, and presented here for those programmers who need to write custom file managers, device drivers, or other extensions to OS-9.

PDs have three sections: the first 10-byte section is defined universally for all file managers and device drivers. The second 22-byte section is reserved for and defined by each type of file manager. The last 32-byte section is used as an "option" area for communications with user programs via the GETSTAT and SETSTAT system calls. The variables in this area are typically used for dynamically-alterable operating parameters for the file or device.

### Universal Path Descriptor Definitions

| Name | Addr | Size | Description |
|---|---|---|---|
| PD.PD | $00 | 1 | Path number |
| PD.MOD | $01 | 1 | Access mode: 1=read 2=write 3=update |
| PD.CNT | $02 | 1 | Number of paths using this PD |
| PD.DEV | $03 | 2 | Address of associated device table entry |
| PD.CPR | $05 | 1 | Requester's process ID |
| PD.RGS | $06 | 2 | Caller's MPU register stack address |
| PD.BUF | $08 | 2 | Address of 256-byte data buffer (if used) |
| PD.FST | $0A | 22 | Reserved for file manager |
| PD.OPT | $20 | 32 | Reserved for GETSTAT/SETSTAT options |

For more information, please see the sections of this manual on file managers, device descriptors, and writing device drivers.

# RANDOM BLOCK FILE MANAGER

The Random Block File Manager (RBFMAN) is the OS-9 module that supports random-access, block-oriented devices such as disk systems, bubble memory systems, and high-performance tape systems. RBFMAN can handle any number or type of such systems simultaneously. It is a reentrant subroutine package called by IOMAN for I/O service requests to random-access devices. It is responsible for maintaining the logical and physical file structures.

In the course of normal operation, RBFMAN requests allocation and deallocation of 256-byte data buffers; usually one is required for each open file. When physical I/O functions are necessary, RBFMAN directly calls the subroutines in the associated device drivers. All data transfers are performed using 256-byte "blocks". RBFMAN does not know about tracks, cylinders, etc. Instead, it passes the drivers a "logical sector number" ranging from Ø to n-1, where n is the maximum number of sectors on the media. The driver is responsible for translating the logical sector number to actual cylinder/track/sector values.

Because RBFMAN is designed to support a wide range of devices having different performance and storage capacity, it is highly parameter-driven. The physical parameters it uses are stored on the media itself. On disk systems, this information is written on the first few sectors of track number zero. The device drivers also use this information, particularly the physical parameters stored on sector Ø. These parameters are written by the "format" program that initializes and tests the media.

NOTE: IOMAN allocates a 64 byte path descriptor when a path is opened or created, and deallocates the path descriptor after the path is closed.

## LOGICAL AND PHYSICAL DISK ORGANIZATION

### Identification Sector

Logical sector number zero contains a description of the physical and logical characteristics of the volume. These are established by the "format" command program when the media is initialized. the table below gives the OS-9 mnemomic name, byte address, size, and description of each value stored in this sector.

| name | addr | size | description |
|---|---|---|---|
| DD.TOT | $00 | 3 | Total number of sectors on media |
| DD.TKS | $03 | 1 | Number of sectors per track |
| DD.MAP | $04 | 2 | Number of bytes in allocation map |
| DD.BIT | $06 | 2 | Number of sectors per cluster |
| DD.DIR | $08 | 3 | Starting sector of root directory |
| DD.OWN | $0B | 2 | Owner's user number |
| DD.ATT | $0D | 1 | Disk attributes |
| DD.DSK | $0E | 2 | Disk identification (for internal use) |
| DD.FMT | $10 | 1 | Disk format: density, number of sides |
| DD.SPT | $11 | 2 | Number of sectors per track. |
| DD.RES | $13 | 2 | Reserved for future use |
| DD.BT | $15 | 3 | Starting sector of bootstrap file |
| DD.BSZ | $18 | 2 | Size of bootstrap file (in bytes) |
| DD.DAT | $1A | 5 | Time of creation: Y:M:D:H:M |
| DD.NAM | $1F | 32 | Volume name: last char has sign bit set |

### Disk Allocation Map Sector

One sector of the disk is used for the "disk allocation map" that specifies which clusters on the disk are available for allocation of file storage space. The address of this sector is always assigned logical sector 1 by the format program. DD.MAP specifies the number of bytes in this sector which are actually used in the map.

Each bit in the map corresponds to a cluster of sectors on the disk. The number of sectors per cluster is specified by the "DD.BIT" variable in the identification sector, and is always an integral power of two, i.e., 1, 2, 4, 8, 16, etc. There are a maximum of 4096 bits in the map, so media such as double-density double-sided floppy disks and hard disks will use a cluster size of two or more sectors. Each bit is cleared if the corresponding cluster is available for allocation, or set if the sector is already allocated, non-existant, or physically defective. The bitmap is initially built by the format program.

## File Descriptor Sectors

The first sector of every file is called a "file descriptor", which contains the logical and physical description of the file. The table below describes the contents of the descriptor.

| name | addr | size | description |
|---|---|---|---|
| FD.ATT | $0 | 1 | File Attributes: D S PE PW PR E W R |
| FD.OWN | $1 | 2 | Owner's User ID |
| FD.DAT | $3 | 5 | Date Last Modified: Y M D H M |
| FD.LNK | $8 | 1 | Link Count |
| FD.SIZ | $9 | 4 | File Size (number of bytes) |
| FD.DCR | $D | 3 | Date Created: Y M D |
| FD.SEG | $10 | 240 | Segment List: see below |

The attribute byte contains the file permission bits. Bit 7 is set to indicate a directory file, bit 6 indicates a "sharable" file, bit 5 is public execute, bit 4 is public write, etc.

The segment list consists of up to 48 five-byte entries that have the size and address of each block of storage that comprise the file in logical order. Each entry has a three-byte logical sector number of the block, and a two-byte block size (in sectors). The entry following the last segment will be zero.

When a file is created, it initially has no data segments allocated to it. Write operations past the current end-of-file (the first write is always past the end-of-file) cause additional sectors to be allocated to the file. If the file has no segments, it is given an initial segment having the number of sectors specified by the minimum allocation entry in the device descriptor, or the number of sectors requested if greater than the minimum. Subsequent expansions of the file are also generally made in minimum allocation increments. An attempt is made to expand the last segment wherever possible rather than adding a new segment. When the file is closed, unused sectors in the last segment are truncated.

A note about disk allocation: OS-9 attempts to minimize the number of storage segments used in a file. In fact, many files will only have one segment in which case no extra read operations are needed to randomly access any byte on the file. Files can have multiple segments if the free space of the disk becomes very fragmented, or if a file is repeatedly closed, then opened and expanded at some later time. This can be avoided by writing a byte at the highest address to be used on a file before writing any other data.

## Directories

Disk directories are files that have the "D" bit set in their attribute byte. Each directory entry is 32 bytes long, consisting of 29 bytes for the file name followed by a three byte logical sector number of the file's descriptor sector. The file name is left-justified in the field with the sign bit of the last character set. Unused entries have a zero byte in the first file name character position. Every mass-storage media must have at least one master directory called the "root directory". The beginning logical sector number of this directory is stored in the identification sector, as previously described.

## RBFMAN Definitions of the Path Descriptor

The table below describes the usage of the file-manager-reserved section of path descriptors used by RBFMAN. Also see the section of this manual on path descriptors.

| Name | Addr | Size | Description |
|---|---|---|---|
| **Universal Section** | | | |
| PD.PD | $00 | 1 | Path number |
| PD.MOD | $01 | 1 | Mode (read/write/update) |
| PD.CNT | $02 | 1 | Number of open images |
| PD.DEV | $03 | 2 | Address of device table entry |
| PD.CPR | $05 | 1 | Current process ID |
| PD.RGS | $06 | 2 | Address of callers register stack |
| PD.BUF | $08 | 2 | Buffer address |
| **RBFMAN Path Descriptor Definitions** | | | |
| PD.SMF | $0A | 1 | State flags (see next page) |
| PD.CP | $0B | 4 | Current logical file position (byte addr) |
| PD.SIZ | $0F | 4 | File size |
| PD.SBL | $13 | 3 | Segment beginning logical sector number |
| PD.SBP | $16 | 3 | Segment beginning physical sector number |
| PD.SSZ | $19 | 2 | Segment size |
| PD.DSK | $1B | 2 | Disk ID (for internal use only) |
| PD.DTB | $1D | 2 | Address of drive table |

(continued)

RBFMAN Option Section Definitions

| | | | |
|---|---|---|---|
| | $20 | 1 | Device class 0= SCF 1=RBF 2=PIPE 3=SBF |
| PD.DRV | $21 | 1 | Drive number (0..N) |
| PD.STP | $22 | 1 | Step rate |
| PD.TYP | $23 | 1 | Device type |
| PD.DNS | $24 | 1 | Density capability |
| PD.CYL | $25 | 2 | Number of cylinders (tracks) |
| PD.SID | $27 | 1 | Number of sides (surfaces) |
| PD.VFY | $28 | 1 | 0 = verify disk writes |
| PD.SCT | $29 | 2 | Default number of sectors/track |
| PD.T0S | $2B | 2 | Default number of sectors/track (track 0) |
| PD.ILV | $2D | 1 | Sector intreleave factor |
| PD.SAS | $2E | 1 | Segment allocation size |

(the following values are NOT copied from the device descriptor)

| | | | |
|---|---|---|---|
| PD.ATT | $33 | 1 | File attributes (D S PE PW PR E W R) |
| PD.FD | $34 | 3 | File descriptor PSN (physical sector #) |
| PD.DFD | $37 | 3 | Directory file descriptor PSN |
| PD.DCP | $3A | 4 | File's directory entry pointer |
| PD.DVT | $3E | 2 | Address of device table entry |

State Flag (PD.SMF): the bits of this byte are defined as:
bit 0 = set if current buffer has been altered
bit 1 = set if current sector is in buffer
bit 2 = set if descriptor sector in buffer

The first section is universal for all file managers, the second and third sections are specific for RBFMAN and RBFMAN-type device drivers. The option section of the path descriptor contains many device operating parameters which may be read and/or written by the OS9 I$GSTT and I$SSTT service requests. This section is initialized by IOMAN who copies the initialization table of the device descriptor into the option section of the path descriptor when a path to a device is opened. Any values not determined by this table will default to zero.

NOTE: For a description of the values copied into the option section of the path descriptor and other related information, please see the section of this manual on device descriptors.

# SEQUENTIAL CHARACTER FILE MANAGER

The Sequential Character File Manager (SCFMAN) is the OS-9 module that supports character-oriented devices such as terminals, printers, modems, etc. SCFMAN can handle any number or type of such systems. It is a reentrant subroutine package called by IOMAN for I/O service requests to sequential character oriented devices. It includes the extensive input and output editing functions typical of line-oriented operation such as: backspace, line delete, repeat line, auto line feed, screen pause, return delay padding, etc.

## SCFMAN Definitions of The Path Descriptor

The table below describes the path descriptors used by SCFMAN and SCFMAN-type device drivers. Also see the section of this manual on path descriptors and device descriptors.

| Name | Offset | Size | Description |
|---|---|---|---|
| **Universal Section** | | | |
| PD.PD | $00 | 1 | Path number |
| PD.MOD | $01 | 1 | Mode (read/write/update) |
| PD.CNT | $02 | 1 | Number of open images |
| PD.DEV | $03 | 2 | Address of device table entry |
| PD.CPR | $05 | 1 | Current process ID |
| PD.RGS | $06 | 2 | Address of callers MPU register stack |
| PD.BUF | $08 | 2 | Buffer address |
| **SCFMAN Path Descriptor Definitions** | | | |
| PD.DV2 | $0A | 2 | Device table addr of 2nd (echo) device |
| PD.RAW | $0C | 1 | Edit flag: 0=raw mode, 1=edit mode |
| PD.MAX | $0D | 2 | Readline maximum character count |
| PD.MIN | $0F | 1 | Devices are "mine" if cleared |

(continued)

SCFMAN Option Section Definition

| | | | |
|---|---|---|---|
| | $20 | 1 | Device class Ø=SCF 1=RBF 2=PIPE 3=SBF |
| PD.UPC | $21 | 1 | Case (Ø=BOTH, 1=UPPER ONLY) |
| PD.BSO | $22 | 1 | Backsp (Ø=BSE, 1=BSE SP BSE) |
| PD.DLO | $23 | 1 | Delete (Ø = BSE over line, 1=CR LF) |
| PD.EKO | $24 | 1 | Echo (Ø=no echo) |
| PD.ALF | $25 | 1 | Auto LF (Ø=no auto LF) |
| PD.NUL | $26 | 1 | End of line null count |
| PD.PAU | $27 | 1 | Pause (Ø= no end of page pause) |
| PD.PAG | $28 | 1 | Lines per page |
| PD.BSP | $29 | 1 | Backspace character |
| PD.DEL | $2A | 1 | Delete line character |
| PD.EOR | $2B | 1 | End of record character (read only) |
| PD.EOF | $2C | 1 | End of file character (read only) |
| PD.RPR | $2D | 1 | Reprint line character |
| PD.DUP | $2E | 1 | Duplicate last line character |
| PD.PSC | $2F | 1 | Pause character |
| PD.INT | $3Ø | 1 | Keyboard interrupt character (CTL C) |
| PD.QUT | $31 | 1 | Keyboard abort character (CTL Q) |
| PD.BSE | $32 | 1 | Backspace echo character (BSE) |
| PD.OVF | $33 | 1 | Line overflow character (bell) |
| PD.PAR | $34 | 1 | Device initialization value (parity) |
| PD.BAU | $35 | 1 | Software settable baud rate |
| PD.D2P | $36 | 2 | Offset to 2nd device name string |
| | $38 | 2 | Reserved for future use |
| PD.ERR | $3A | 1 | Accumulated I/O error status. |
| PD.TBL | $3B | 2 | Address of device table |

The first section is universal for all file managers, the second and third section are specific for SCFMAN and SCFMAN-type device drivers. The option section of the path descriptor contains many device operating parameters which may be read or written by the OS9 I$GSTT or I$SSTT service requests. IOMAN initializes this section when a path is opened to a device by copying the corresponding device descriptor initialization table. Any values not determined by this table will default to zero.

Special editing functions may be disabled by setting the corresponding control character value to zero.

## SCFMAN Line Editing Features

I$READ and I$WRITE service requests to SCFMAN type devices generally pass the data to/from the device without any modification, except that keyboard interrupt, keyboard abort, and pause character are filtered out of the input (editing is disabled if the corresponding character in the path descriptor contains a zero). In particular carriage returns are not automatically followed by line feeds or nulls, and the high order bits are passed as sent/received.

I$RDLN and I$WRLN service requests to SCFMAN type devices will cause the following editing to occur:

All bytes input or output have their high order bit cleared.

If PD.UPC <> Ø bytes input or output in the range "a..z" are made "A..Z"

If PD.EKO <> Ø, input bytes are echoed, except that undefined control characters in the range $Ø..$1F print as "."

If PD.ALF <> Ø, carriage returns are automatically followed by line feeds.

If PD.NUL <> Ø, After each CR/LF a PD.NUL "nulls" (always $ØØ) are sent.

If PD.PAU <> Ø, Auto page pause will occur after every PD.PAU lines since the last input.

If PD.BSP <> Ø, SCF will recognize PD.BSP as the "input" backspace character, and will echo PD.BSE (the backspace echo character) if PD.BSO = Ø, or PD.BSE, space, PD.BSE if PD.BSO <> Ø.

If PD.DEL <> Ø, SCF will recognize PD.DEL the delete line character (on input), and echo the "backspace sequence over the entire line if PD.DLO = Ø, or echo CR/LF if PD.DLO <> Ø

PD.EOR defines the end of record character. This is the last character on each line entered (I$RDLN), and terminates the output (I$WRLN) when this character is sent. Normally PD.EOR will be set to $ØD. If it is set to zero, SCF's READLN will NEVER terminate, unless an EOF occurs.

If PD.EOF <> Ø, it defines the end of file character. SCFMAN will return an end-of-file error on I$READ or I$RDLN if this is the first (and only) character input. It can be disabled by setting its value to zero.

If PD.RPR <> Ø, SCF (I$RDLN) will, upon receipt of this character, echo a carriage return [optional line feed], and then reprint the current line.

If PD.DUP <> Ø, SCF (I$RDLN) will duplicate whatever is in the input buffer through the first "PD.EOR" character.

If PD.PSC <> Ø, output is suspened before the next "PD.EOR" character when this character is input. This will also delete any "type ahead" input for I$RDLN.

If PD.INT <> Ø, and it is received on input, a keyboard interrupt signal is sent to the last user of this path. Also it will terminate the current I/O request (if any) with an error identical to the keyboard interrupt signal code. This location normally is set to a control-C character.

If PD.QUT <> Ø, and it is received on input, a keyboard abort signal is sent to the last user of this path. Also it will terminate the current I/O request (if any) with an error code identical to the keyboard interrrupt signal code. This location is normally set to a control-Q character.

If PD.OVF <> Ø, It is echoed when I$RDLN has satisfied its input byte count without finding a "PD.EOR" character.

NOTE: It is possible to disable most of these special editing functions by setting the corresponding control character in the path descriptor to zero by using the I$SSTT service request. A more "permanent" solution may be had by setting the corresponding control character value in the device descriptor to zero.

Device descriptors may be inspected to determine the default settings for these values.

# INTERRUPT PROCESSING

The OS-9 kernel ROMS contain the hardware vectors required by the 6809 MPU at addresses $FFF0 through $FFFF. These vectors each point to jump-extended-indirect instruction which vector the MPU to the actual interrupt service routine. A RAM vector table in page zero of memory contains the target addresses of the jump instructions as follows:

| INTERRUPT | ADDRESS |
|---|---|
| SWI3 | $002C |
| SWI2 | $002E |
| FIRQ | $0030 |
| IRQ | $0032 |
| SWI | $0034 |
| NMI | $0036 |

OS-9 initializes each of these locations after reset to point to a specific service routine in the kernel. The SWI, SWI2, and SWI3 vectors point to specific routines which in turn read the corresponding pseudo vector from the process' process descriptor and dispatch to it. This is why the F$SSWI service request to be local to a process since it only changes a pseudo vector in the process descriptor. The IRQ routine points directly to the IRQ polling system, or to it indirectly via the real-time clock device service routine. The FIRQ and NMI vectors are not normally used by OS-9 and point to RTI instructions.

A secondary vector table located at $FFE0 contains the addresses of the routines that the RAM vectors are initialized to. They may be used when it is necessary to restore the original service routines after altering the RAM vectors. On the next page are the definitions of both the actual hardware interrupt vector table, and the secondary vector table:

## OS-9 Interrupt Vector Tables

| VECTOR | ADDRESS | |
|---|---|---|
| **OS-9 Secondary Vector Table** | | |
| TICK | $FFE0 | Clock Tick Service Routine |
| SWI3 | $FFE2 | |
| SWI2 | $FFE4 | |
| FIRQ | $FFE6 | |
| IRQ | $FFE8 | |
| SWI | $FFEA | |
| NMI | $FFEC | |
| WARM | $FFEE | Reserved for warm-start |
| **Actual Hardware Vector Table** | | |
| SWI3 | $FFF2 | |
| SWI2 | $FFF4 | |
| FIRQ | $FFF6 | |
| IRQ | $FFF8 | |
| SWI | $FFFA | |
| NMI | $FFFC | |
| RESTART | $FFFE | |

If it is necessary to alter the RAM vectors use the secondary vector table to exit the substitute routine. The technique of altering the IRQ pointer is usually used by the clock service routines to reduce latency time of this frequent interrupt source.

## IRQ AUTOMATIC POLLING SYSTEM

In OS-9 systems, most I/O devices use IRQ-type interrupts, so OS-9 includes a sophisticated polling system that automatically identifies the source of the interrupt and dispatches to its associated user-defined service routine. The information required for IRQ polling is maintained in a data structure called the "IRQ polling table". The table has a 9-byte entry for each possible IRQ-generating device. The table size is static and defined by an initialization constant in the System Configuration Module.

The polling system is prioritized so devices having a relatively greater importance (i.e., interrupt frequency) are polled before those of lesser priority. This is accomplished by keeping the entries sorted by priority, which is a number between Ø (lowest) and 255 (highest). Each entry in the table has 6 variables:

1. POLLING ADDRESS: The address of the device's status register, which must have a bit or bits that indicate it is the source of an interrupt.

2. MASK BYTE: This byte selects one or more bits within the device status register that are interrupt request flag(s). A set bit identifies the active bit(s).

3. FLIP BYTE: This byte selects whether the bits in the device status register are true when set or true when cleared. Cleared bits indicate active when set.

4. SERVICE ROUTINE ADDRESS: The user-supplied address of the device's interrupt service routine.

5. STATIC STORAGE ADDRESS: a user-supplied pointer to the permanent storage required by the device service routine.

6. PRIORITY: The device priority number: Ø to 255. This value determines the order in which the devices in the polling table will be polled. Note: this is not the same as a process priority which is used by the execution scheduler to decide which process gets the next time slice for MPU execution.

When an IRQ interrupt occurs, the polling system is entered via the corresponding RAM interrupt vector. It starts polling the devices, using the entries in the polling table in priority order. For each entry, the status register address is loaded into accumulator A using the device address from the table. An

exclusive-or operation using the "flip-byte" is executed, followed by a logical-and operation using the mask byte. If the result is non-zero, the device is assumed to be the cause of the interrupt. The device's static storage address and service routine address is read from the table, and jumped to.

NOTE: The interrupt service routine should terminate with an RTS - NOT AN RTI instruction.

Entries can be made to the IRQ polling table by use of a special OS-9 service request called "F$IRQ". This is a priviledged service request that can be executed only when OS-9 is in System Mode (which is the case when device drivers are executed).

## WRITING INTERRUPT-DRIVEN DEVICE DRIVERS

It is important to understand that interrupt service routines are asynchronous and somewhat nebulous in that they are not distinct processes. In fact, when they are invoked ANY indeterminate process may have been interrupted, but not necessarily the process that triggered the interrupt-causing event.

Therefore, all interrupt-driven device drivers have two basic parts: the "mainline" subroutines that execute as part of the calling process, and a separate interrupt service routine.

THE TWO ROUTINES ARE ASYNCHRONOUS AND THEREFORE MUST USE SIGNALS FOR COMMUNICATIONS AND COORDINATION.

The INIT initialization subroutine within the driver package should allocate static storage for the service routine, get the service routine address, and execute the F$IRQ system call to add it to the IRQ polling table.

When a device driver routine does something that will result in an interrupt, it should immediately execute a F$SLEP service request. This results in the process' deactivation. When the interrupt in question occurs, its service routine is executed after some random interval. It should then do the minimal amount of processing required, and send a "wakeup" signal to its associated process using the F$SEND service request. It may also put some data in its static storage (I/O data and status) which is shared with its associated "sleeping" process.

Some time later, the device driver "mainline" routine is awakened by the signal, and can process the data or status returned by the interrupt service routine. Remember that processes that execute "sleep" requests while in system state are given maximum priority by the scheduler.

# WRITING OS-9 ASSEMBLY LANGUAGE PROGRAMS

There are four key rules for programmers writing OS-9 assembly language programs:

1. All programs MUST use position-independent-code (PIC). OS-9 selects load addresses based on available memory at run-time. There is no way to force a program to be loaded at a specific address.

2. Programs must be organized as contiguous memory modules, which are the OS-9 equivalent of "load records".

3. Storage for variables and data structures must be part of the data area which is assigned by OS-9 at run-time, and is separate from the program module (section).

4. All input and output operations should be made using OS-9 service request calls.

Fortunately, the 6809's versatile addressing modes make the rules above easy to follow. The OS-9 assembler also helps because it has special capabilities to assist the programmer in creating programs for the OS-9 execution environment.

## Using Position-Independent Code

It is simple to write 6809 Position-independent-code (PIC). The trick is to always use PC-relative addressing; for example BRA, LBRA, BSR and LBSR. Get addresses of constants and tables using LEA instructions instead of load immediate instructions. If you use dispatch tables, use tables of RELATIVE, not absolute, addresses.

| INCORRECT | CORRECT |
|---|---|
| LDX #TABLE | LEAX TABLE,PCR |
| JSR SUBR | BSR SUBR or LBSR SUBR |

## Standard Input/Output Paths

Programs should be written to use standard I/O paths #0 (input), #1 (output), and #2 (error output) wherever practical. Usually, this involves I/O calls that are intended to communicate to the user's terminal, or any other case where the OS-9 redirected I/O capability is desirable. These paths do not have to be OPENed or CLOSEd.

## How to Select Addressing Modes

Programs to be invoked using FORK and CHAIN system calls have RAM memory assigned at execution-time. The addresses cannot be known or specified ahead of time. Again, thanks to the 6809's full compliment of addressing modes this presents no problem.

When the program is first entered, the Y register will have the address of the top of your data memory area. If the creating process passed a parameter area, it will be located from the value of the SP to the top of memory (Y), and the D register will contain the parameter area size in bytes. If the new process was called by the shell, the parameter area will contain the part of the shell command line that includes the argument (parameter) text. The U register will have the lower bound of the data memory area, and the DP register will contain its page number.

The most important rule is to NOT USE EXTENDED ADDRESSING! Indexed and direct page addressing should be used exclusively to access data area values and structures. Do not use program-counter relative addressing to find addresses in the data area, but do use it to refer to addresses within the program area.

The most efficient way to handle tables, buffers, stacks, etc., is to have the program's initialization routine compute their absolute addresses using the data area bounds passed by OS-9 in the registers. These addresses can then be saved in the direct page where they can be loaded into registers quickly, using short instructions.

This technique has advantages: it is faster than extended addressing, and the program is inherently reentrant.

## Machine Stack Requirements

Because OS-9 uses interrupts extensively, and also because many reentrant 6809 programs use the MPU stack for local variable storage, a generous stack should be maintained at all times. The recommended minimum is approximately 200 bytes.

## Interrupt Masks

User programs should keep the condition codes register F (FIRQ mask) and I (IRQ mask) bits off. They can be set during critical program sequences to avoid task-switching or interrupts, but this time should be kept to a mimimum. If they are set for longer than a tick period, system timekeeping accuracy will be affected.

Example Program

The example program shown below is presented here to provide some idea as to how addressing modes etc. are actually used. This program will print "HELLO WORLD" on the terminal, and then wait for a line to be typed in.

```
         NAM    EXAMPLE
         OPT    -M
         USE    /D0/DEFS/OS9DEFS
*
* OS-9 System Definition File Included
*

         ORG    0
LINLEN   RMB    2              SAVE LINE LENGTH HERE
INPBUF   RMB    80             LINE INPUT BUFFER
         RMB    200            MINIMUM HARDWARE STACK SIZE
STACK    EQU    .-1
DATMEM   EQU    .              DATA AREA MEMORY SIZE

         MOD    ENDPGM,NAME,OBJCT+PRGRM,REENT+1,ENTRY,DATMEM

NAME     FCS    /EXAMPLE/      MODULE NAME STRING
ENTRY    EQU    *              MODULE EXECUTION ENTRY POINT
         LEAX   OUTSTR,PCR     GET ADDRESS OF OUTPUT STRING
         LDY    #STRLEN        GET STRING LENGTH
         LDA    #1             GET STANDARD OUTPUT PATH NUMB
         OS9    I$WRLN         WRITE THE LINE
         BCS    ERROR          BRA IF ANY I/O ERRORS OCCURED
         LEAX   INPBUF,U       GET ADDR OF INPUT BUFFER
         LDY    #80            INPUT MAX OF 80 CHARACTERS
         LDA    #0             GET STANDARD INPUT PATH NUMBE
         OS9    I$RDLN         READ THE LINE INTO THE BUFFER
         BCS    ERROR          BRA IF ANY I/O ERRORS OCCURED
         STY    LINLEN         SAVE THE LINE LENGTH
         CLRB                  RETURN WITH NO ERRORS
ERROR    OS9    F$EXIT         TERMINATE THE PROCESS

OUTSTR   FCC    /HELLO WORLD/ OUTPUT STRING
         FCB    $0D            END OF LINE CHARACTER
STRLEN   EQU    *-OUTSTR       STRING LENGTH

         EMOD                  END OF MODULE
ENDPGM   EQU    *              END OF PROGRAM
```

NOTE: The OS9DEFS system definitions file is supplied with the OS-9 assembler. Also this file may be located in a different directory than the one given in the "USE" statement in the program.

# OS-9 SERVICE REQUEST DESCRIPTIONS

System calls are used to communicate between the OS-9 operating system and assembly-language-level programs. There are three general categories:

1. User mode function requests
2. System mode function requests
3. I/O requests

System mode fuction requests are priviledged and may be executed only while OS-9 is in the system state (when it is processing another service request). They are included in this manual primarily for the benefit of those programmers who will be writing device drivers and other sophisticated applications.

The system calls are performed by loading the MPU registers with the appropriate parameters (if any), and executing a SWI2 instruction immediately followed by a constant byte which is the request code. Parameters (if any) will be returned in the MPU registers after OS-9 has processed the service request. A standard convention for reporting errors is used in all system calls; if an error occured, the "C bit" of the condition code register will be set and accumulator B will contain the appropriate error code. This permits a BCS or BCC instruction immediately following the system call to branch on error/no error.

Here's an example system call for the "CLOSE" service request (code $8B):

```
LDA PATHNUM
SWI2
FCB $8B
BCS ERROR
```

Using the assembler's "OS9" directive simplifies the call:

```
LDA PATHNUM
OS9 I$CLOS
BCS ERROR
```

The I/O service requests are simpler to use than in many other operating systems because the calling program does not have to allocate and set up "file control blocks", "sector buffers", etc. Instead OS-9 will return a one byte path number when a path to a file/device is opened or created; then this path number may be used in subsequent I/O requests to identify the file/device until the path is closed. OS-9 internally allocates and maintains its own data structures and users never have to deal with them: in fact attempts to do so are memory violations.

All system calls have a mnemonic name that starts with "F$" for system functions, or "I$" for I/O related requests. These are defined in the assembler-input equate file called "OS9DEFS".

In the service request descriptions which follow, registers not explicitly specified as input or output parameters are not altered.

Strings passed as parameters are normally terminated by having bit seven of the last character set, a space character, or an end of line character.

ABIT Set bits in an allocation bit map F$ABIT

ASSEMBLER CALL: OS9 F$ABIT

MACHINE CODE: 103F 13

INPUT: (X) = Base address of allocation bit map.
(D) = Bit number of first bit to set.
(Y) = Bit count (number of bits to set).

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request sets bits in the allocation bit map specified by the X register.

Bit numbers range from 0..N-1, where N is the number of bits in the allocation bit map.

CHAIN     Load and execute a new primary module.     F$CHAN

ASSEMBLER CALL: OS9 F$CHAN

MACHINE CODE: 103F 05

INPUT: (X) = Address of module name or file name.
(Y) = Parameter area size (256 byte pages).
(U) = Beginning address of parameter area.
(A) = Language / type code.
(B) = Optional data area size (256 byte pages).

ERROR OUPTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call is similar to FORK, but it does not create a new process. It effectively "resets" the calling process' program and data memory areas and begins execution of a new primary module. Open paths are not closed or otherwise affected.

This system call is used when it is necessary to execute an entirely new program, but without the overhead of creating a new process. It is functionally similar to a FORK followed by an EXIT, but with less processing overhead.

The sequence of operations taken by CHAIN is as follows:

1. The system parses the name string of the new process' "primary module" - the program that will initially be executed. Then the system module directory is searched to see if a module with the same name and type / language is already in memory. If so it is linked to. If not, the name string is used as the pathlist of a file which is to be loaded into memory. Then the first module in this file is linked to (several modules may have been loaded from a single file).

2. The process' old primary module is UNLINKED.

3. The data memory area is reconfigured to the size specified in the new primary module's header.

CHAIN (continued)

The diagram below shows how CHAIN sets up the data memory area and registers for the new module.

```
+-----------------+  <-- Y
|                 |
|   parameter     |
|     area        |
|                 |
+-----------------+  <-- X, SP
|                 |
|                 |
|   data area     |
|                 |
|                 |
+-----------------+
|  direct page    |
+-----------------+  <--  U, DP

  D = parameter area size
 PC = module entry point abs. address
 CC = F=Ø, I=Ø, others undefined
```

Y (top of memory pointer) and U (bottom of memory pointer) will always have a values at 256-byte page boundaries. If the parent does not specify a parameter area, Y, X, and SP will be the same, and D will equal zero. The minimum overall data area size is one page (256 bytes).

WARNING: The hardware stack pointer (SP) should be located somewhere in the direct page before the F$CHAN service request is executed to prevent a "suicide attempt" error or an acutal suicide (system crash). This will prevent a suicide from occuring in case the new module requires a smaller data area than what is currently being used. You should allow approximately 2ØØ bytes of stack space for execution of the F$CHAN service request and other system "overhead".

For more information, please see the F$FORK service request description.

COMPARE NAMES    Compare two names.    F$CNAM

ASSEMBLER CALL: OS9 F$CNAM

MACHINE CODE: 103F 11

INPUT: (X) = Address of first name.
(B) = Length of first name.
(Y) = Address of second name.

OUTPUT: (CC) = C bit clear if the strings match.

Given the address and length of a string, and the address of a second string, compares them and indicates whether they match. Typically used in conjunction with "parsename".

The second name must have the sign bit (bit 7) of the last character set.

CRC Compute CRC F$CRC

ASSEMBLER CALL: OS9 F$CRC

MACHINE CODE: 103F 17

INPUT: (X) = Starting byte address.
(Y) = Byte count.
(U) = Address of 3 byte CRC accumulator.

OUTPUT: CRC accumulator is updated.

ERROR OUTPUT: None.

This service request calculates the CRC (cyclic redundancy count) for use by compilers, assemblers, or other module generators. The CRC is calculated starting at the source address over "byte count" bytes. It is not necessary to cover an entire module in one call, since the CRC may be "accumulated" over several calls. The CRC accumulator must be initialized to $FFFFFF before the first F$CRC call.

The last three bytes in the module MUST be $FFFFFF. This is the initial module CRC and is normally replaced by the calculated value.

DBIT Deallocate in a bit map F$DBIT

ASSEMBLER CALL: OS9 F$DBIT

MACHINE CODE: 103F 14

INPUT: (X) = Base address of an allocation bit map.
(D) = Bit number of first bit to clear.
(Y) = Bit count (number of bits to clear).

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request is used to clear bits in the allocation bit map pointed to by X.

Bit numbers range from Ø..N-1, where N is the number of bits in the allocation bit map.

EXIT Terminate the calling process. F$EXIT

ASSEMBLER CALL: OS9 F$EXIT

MACHINE CODE: 103F 06

INPUT: (B) = Status code to be returned to the parent process.

OUTPUT: Process is terminated.

This call kills the calling process. Its data memory area is deallocated, and its primary module is UNLINKed. All open paths are automatically closed.

The death of the process can be detected by the parent executing a WAIT call, which returns to the parent the status byte passed by the child in its EXIT call. The status byte can be an OS-9 error code the terminating process wishes to pass back to its parent process (the shell assumes this), or can be used to pass a user-defined status value. Processes to be called directly by the shell should only return an OS-9 error code or zero if no error occurred.

FORK Create a new process. F$FORK

ASSEMBLER CALL: OS9 F$FORK

MACHINE CODE: 103F 03

INPUT: (X) = Address of module name or file name.
(Y) = Parameter area size.
(U) = Beginning address of the parameter area.
(A) = Language / Type code.
(B) = Optional data area size (pages).

OUTPUT: (X) = Updated path the name string.
(A) = New process ID number.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call creates a new process which becomes a "child" of the caller, and sets up the new process' memory and MPU registers.

The system parses the name string of the new process' "primary module" - the program that will initially be executed. Then the system module directory is searched to see if the program is already in memory. If so, the module is linked to and executed. If not, the name string is used as the pathlist of the file which is to be loaded into memory. Then the first module in this file is linked to and executed (several modules may have been loaded from a single file).

The primary module's module header is used to determine the process' initial data area size. OS-9 then attempts to allocate a contiguous RAM area equal to the required data storage size, (includes the parameter passing area, which is copied from the parent process' data area). The new process' registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module's entry point.

(continued)

FORK (continued)

When the shell processes a command line it passes a string in the parameter area which is a copy of the parameter part (if any) of the command line. It also inserts an end-of-line character at the end of the parameter string to simplify string-oriented processing. The X register will point to the beginning of the parameter string. If the command line included the optional memory size specification (#n or #nK), the shell will pass that size as the requested memory size when executing the FORK.

If any of the above operations are unsucessful, the FORK is aborted and the caller is returned an error.

The diagram below shows how FORK sets up the data memory area and registers for a newly-created process.

```
+-----------------+  <-- Y
|                 |
|   parameter     |
|     area        |
|                 |
+-----------------+  <-- X, SP
|                 |
|                 |
|    data area    |
|                 |
|                 |
+-----------------+
|  direct page    |
+-----------------+  <--  U, DP

  D = parameter area size
 PC = module entry point abs. address
 CC = F=0, I=0, others undefined
```

Y (top of memory pointer) and U (bottom of memory pointer) will always have a values at 256-byte page boundaries. If the parent does not specify a parameter area, Y, X, and SP will be the same, and D will equal zero. The minimum overall data area size is one page (256 bytes). Shell will always pass at least an end of line character in the parameter area.

INTERCEPT          Set up a signal intercept trap.          F$ICPT

ASSEMBLER CALL:  OS9  F$ICPT

MACHINE CODE:  103F 09

INPUT:  (X) = Address of the intercept routine.
(U) = Address of the intercept routine local storage.

OUTPUT:  None.

ERROR OUTPUT:  (CC) = C bit set.
(B) = Appropriate error code.

This system call tells OS-9 to set a signal intercept trap, where X contains the adddress of the signal handler routine, and U contains the base address of the routine's storage area. After a signal trap has been set, whenever the process receives a signal, its intercept routine will be executed. A signal will abort any process which has not used the F$ICPT service request to set a signal trap, and its termination status (B register) will be the signal code. Many interactive programs will set up an intercept routine to handle keyboard abort (control Q), and keyboard interrupt (control C).

The intercept routine is entered asynchronously because a signal may be sent at any time (it is like an interrupt) and is passed the following:

U = Address of intercept routine local storage.
B = Signal code.

NOTE: The value of DP may not be the same as it was when the F$ICPT call was made.

Whenever a signal is received, OS-9 will pass the signal code and the base address of its data area (which was defined by a F$ICPT service request) to the signal intercept routine. The base address of the data area is selected by the user and is typically a pointer to the process' data area.

The intercept routine is activated when a signal is received, then it takes some action based upon the value of the signal code such as setting a flag in the process' data area. After the signal has been processed, the handler routine should terminate with an RTI instruction.

GET ID          Get process ID / user ID          F$ID

ASSEMBLER CALL: OS9 F$ID

MACHINE CODE: 103F ØC

INPUT: Nothing.

OUTPUT: (A) = Process ID.
(Y) = User ID.

ERROR OUTPUT: (CC) = C Bit set.
(B) = Appropriate error code.

Returns the caller's process ID number, which is a byte value in the range of 1 to 255, and the user ID which is a integer in the range Ø to 65535. The process ID is assigned by OS-9 and is unique to the process. The user ID is defined in the system password file, and is used by the file security system and a few other functions. Several processes can have the same user ID.

LINK: Link to memory module. F$LINK

ASSEMBLER CALL: OS9 F$LINK

MACHINE CODE: 103F 00

INPUT: (X) = Address of the module name string.
(A) = Module type / language byte.

OUTPUT: (X) = Advanced past the module name.
(Y) = Module entry point absolute address.
(U) = Module header absolute address.
(A) = Module type / language.
(B) = Module attributes / revision level.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropritate error code.

This system call causes OS-9 to search the module directory for a module having a name, language and type as given in the parameters. If found, the address of the module's header is returned in U, and the absolute address of the module's execution entry point is returned in Y (as a convenience: this and other information can be obtained from the module header). The module's "link count" is incremented whenever a LINK references its name, thus keeping track of how many processes are using the module. If the module requested has an attribute byte indicating it is not sharable (meaning it is not reentrant) only one process may link to it at a time.

Possible errors:

(A) Module not found.
(B) Module busy (not sharable and in use).
(C) Incorrect or defective module header.

LOAD    Load module(s) from a file.    F$LOAD

ASSEMBLER CALL: OS9 F$LOAD

MACHINE CODE: 103F 01

INPUT: (X) = Address of pathlist (file name)
(A) = Language / type (Ø = any language / type)

OUTPUT: (X) = Advanced past pathlist
(Y) = Primary module entry point address
(U) = Address of module header
(A) = Language / type
(B) = Attributes / revision level

ERROR OUTPUT: (CC) = C Bit set
(B) = Appropriate error code

Opens a file specified by the pathlist, reads one or more memory modules from the file into memory, then closes the file. All modules loaded are added to the system module directory, and the first module read is LINKed. The parameters returned are the same as the LINK call and apply only to the first module loaded.

In order to be loaded, the file must have the "execute" permission and contain a module or modules that have a proper module header. The file will be loaded from the working execution directory unless the pathlist specifies otherwise.

Possible errors: module directory full; memory full; plus errors that occur on OPEN, READ, CLOSE and LINK system calls.

MEM  Resize data memory area.  F$MEM

ASSEMBLER CALL: OS9 F$MEM

MACHINE CODE: 103F 07

INPUT: (D) = Desired new memory area size in bytes.

OUTPUT: (Y) = Address of new memory area upper bound.
(D) = Actual new memory area size in bytes.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Used to expand or contract the process' data memory area. The new size requested is rounded up to the next 256-byte page boundary. Additional memory is allocated contiguously upward (towards higher addresses), or deallocated downward from the old highest address. If D = 0, then the current upper bound and size will be returned.

This request can never return all of a process' memory, or the page in which its SP register points to.

In Level One systems, the request may return an error upon an expansion request even though adequate free memory exists. This is because the data area is always made contiguous, and memory requests by other processes may fragment free memory into smaller, scattered blocks that are not adjacent to the caller's present data area. Level Two systems do not have this restriction because of the availability of hardware for memory relocation, and because each process has its own "address space".

PRERR Print error message. F$PERR

ASSEMBLER CALL: OS9 F$PERR

MACHINE CODE: 103F 0F

INPUT: (A) = Output path number.
(B) = Error code.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This is the system's error reporting utility. It writes an error message to the output path specified. Most OS-9 systems will display:

ERROR #<decimal number>

by default. The error reporting routine is vectored and can be replaced with a more elaborate reporting module. To replace this routine use the F$SSVC service request.

PARSENAME — Parse a path name. — F$PNAM

ASSEMBLER CALL: OS9 F$PNAM

MACHINE CODE: 103F 10

INPUT: (X) = Address of the pathlist.

OUTPUT: (X) = Updated past the optional "/"
(Y) = Address of the last character of the name + 1.
(B) = Length of the name.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.
(X) = Updated past space characters.

Parses the input text string for a legal OS-9 name. The name is terminated by any character that is not a legal component character. This system call is useful for processing pathlist arguments passed to new processes. Also if X was at the end of a pathlist, a bad name error will be returned and X will be moved past any space characters so that the next pathlist in a command line may be parsed.

Note that this system call processes only one name, so several calls may be needed to process a pathlist that has more than one name.

BEFORE F$PNAM CALL:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---
! / ! D ! 0 ! / ! F ! I ! L ! E !   !   !   !   !
+---+---+---+---+---+---+---+---+---+---+---+---+---
  ^
  X
```

AFTER THE F$PNAM CALL:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---
! / ! D ! 0 ! / ! F ! I ! L ! E !   !   !   !   !
+---+---+---+---+---+---+---+---+---+---+---+---+---
      ^       ^
      X       Y         (B) = 2
```

SBMAP Search bit map for a free area **F$SBIT**

ASSEMBLER CALL: OS9 F$SBIT

MACHINE CODE: 103F 12

INPUT: (X) = Beginning address of a bit map.
(D) = Beginning bit number.
(Y) = Bit count (free bit block size).
(U) = End of bit map address.

OUTPUT: (D) = Beginning bit number.
(Y) = Bit count.

This system mode service request searches the specified allocation bit map starting at the "beginning bit number" for a free block (cleared bits) of the required length.

If no block of the specified size exists, it returns with the carry set, beginning bit number and size of the largest block.

SEND Send a signal to another process. F$SEND

ASSEMBLER CALL: OS9 F$SEND

MACHINE CODE: 103F 08

INPUT: (A) = Reciever's process ID number.
(B) = Signal code.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call sends a "signal" to the process specified. The signal code is a single byte value of 1 - 255.

If the signal's destination process is sleeping or waiting, it will be activated so that it may process the signal. The signal processing routine (intercept) will be executed if a signal trap was set up (see F$ICPT), otherwise the signal will abort the destination process, and the signal code becomes the exit status (see WAIT). An exception is the WAKEUP signal, which activates a sleeping process but does not cause the signal intercept routine to be executed.

Some of the signal codes have meanings defined by convention:

```
      0 = System Abort (cannot be intercepted)
      1 = Wake Up Process
      2 = Keyboard Abort
      3 = Keyboard Interrupt
  4-255 = user defined
```

If an attempt is made to send a signal to a process that has an unprocessed, previous signal pending, the current "send" request will be cancelled and an error will be returned. An attempt can be made to resend the signal later. It is good practice to issue a "sleep" call for a few ticks before a retry to avoid wasting MPU time.

For related information see the F$ICPT, F$WAIT, and F$SLEP service request descriptions.

SLEEP Put calling process to sleep. F$SLEP

ASSEMBLER CALL: OS9 F$SLEP

MACHINE CODE: 103F 0A

INPUT: (X) = Sleep time in ticks (0 = indefinitely)

OUTPUT: (X) = Decremented by the number of ticks that the process was asleep.

ERROR OUTPUT: (CC) = C bit set
(B) = Appropriate error code.

This call deactivates the calling process for a specified time, or indefinitely if X = 0. The process will be activated before the full time interval if a signal is received, therefore sleeping indefinitely is a good way to wait for a signal without wasting CPU time.

The duration of a "tick" is system dependent but is most commonly 100 milliseconds.

Due to the fact that it is not known when the F$SLEP request was made durring the current tick, F$SLEP can not be used for precise timing. A sleep of one tick is effectively a "give up remaining time slice" request; the process is immediately inserted into the active process queue and will resume execution when it reaches the front of the queue. A sleep of two or more ticks causes the process to be inserted into the active process queue after N-1 ticks occur and will resume execution when it reaches the front of the queue.

SETPR Set process priority. F$SPRI

```
ASSEMBLER CALL:   OS9  F$SPRI

MACHINE CODE:     103F ØD

INPUT:   (A) = Process ID number.
         (B) = Priority:
                 Ø = lowest
               255 = highest

OUTPUT: None.

ERROR OUTPUT:     (CC) = C bit set.
                  (B)  = Appropriate error code.
```

Changes the process's priority to the new value given. $FF is the highest possible priority, $ØØ is the lowest. A process can change another process' priority only if it has the same user ID.

SSVC Install function request F$SSVC

ASSEMBLER CALL: OS9 F$SSVC

ASSEMBLER CODE: 103F 32

INPUT: (Y) = Address of service request initialization table.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request is used to add a new function request to OS-9's user and privileged system service request tables, or to replace an old one. The Y register passes the address of a table which contains the function codes and offsets to the corresponding service request handler routines. This table has the following format:

```
  OFFSET

            +-----------------------+
   $00      !     Function Code     !   <--- First entry
            +-----------------------+
   $01      ! Offset From Byte 3    !
            +--                   --+
   $02      ! To Function Handler   !
            +-----------------------+
   $03      !     Function Code     !   <--- Second entry
            +-----------------------+
   $04      ! Offset From Byte 6    !
            +--                   --+
   $05      ! To Function Handler   !
            +-----------------------+
            !                       !   <--- Third entry etc.
            !   MORE ENTRIES        !
            !                       !
            !                       !
            +-----------------------+
            !        $80            !   <--- End of table mark
            +-----------------------+
```

NOTE: If the sign bit of the function code is set, only the system table will be updated. Otherwise both the system and user tables will be updated. Privileged system service requests may be called only while executing a system routine.

(continued)

SSVC (continued)

The service request handler routine should process the service request and return from subroutine with an RTS instruction. They may alter all MPU registers (except for SP). The U register will pass the address of the register stack to the service request handler as shown in the following diagram:

```
                                    OFFSET   OS9DEFS
                                             NMEMONIC
                +-------+
  U --->        !  CC   !            $0        R$CC
                +-------+            $1        R$D
                !  A    !            $1        R$A
                +-------+
                !  B    !            $2        R$B
                +-------+
                !  DP   !            $3        R$DP
                +-------+-------+
                !       X       !    $4        R$X
                +---------------+
                !       Y       !    $6        R$Y
                +---------------+
                !       U       !    $8        R$U
                +---------------+
                !       PC      !    $A        R$PC
                +---------------+
```

Function request codes are broken into the two categories as shown below:

| | |
|---|---|
| \$00 - \$28 | User mode service request codes. |
| \$29 - \$34 | Privileged system mode service request codes. When installing these service request, the sign bit should be set if it is to be placed into the system table only. |

NOTE: These categories are defined by convention and not enforced by OS9.

Codes \$25..\$28, and \$70..\$7F will not be used by MICROWARE and are free for user definition.

SETSWI Set SWI vector. F$SSWI

ASSEMBLER CALL: OS9 F$SSWI

MACHINE CODE: 103F 0E

INPUT: (A) = SWI type code.
(X) = Address of user SWI service routine.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Sets up the interrupt vectors for SWI, SWI2 and SWI3 instructions. Each process has its own local vectors. Each SETWSI call sets up one type of vector according to the code number passed in A.

1 = SWI
2 = SWI2
3 = SWI3

When a process is created, all three vectors are initialized with the address of the OS-9 service call processor.

WARNING: Microware-supplied software uses SWI2 to call OS-9. If you reset this vector these programs will not work. If you change all three vectors, you will not be able to call OS-9 at all.

SETIME        Set system date and time.        F$STIM

ASSEMBLER CALL: OS9 F$STIM

MACHINE CODE: 103F 16

INPUT: (X) = Address of time packet (see below)

OUTPUT: Time/date is set.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This service request is used to set the current system date/time. The date and time are passed in a time packet as follows:

| OFFSET | VALUE |
|---|---|
| 0 | year |
| 1 | month |
| 2 | day |
| 3 | hours |
| 4 | minutes |
| 5 | seconds |

TIME | Get system date and time. | F$TIME

ASSEMBLER CALL: OS9 F$TIME

MACHINE CODE: 103F 15

INPUT: (X) = Address of place to store the time packet.

OUTPUT: Time packet (see below).

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This returns the current system date and time in the form of a six byte packet (in binary). The packet is copied to the address passed in X. The packet looks like:

| OFFSET | VALUE |
|---|---|
| 0 | year |
| 1 | month |
| 2 | day |
| 3 | hours |
| 4 | minutes |
| 5 | seconds |

UNLINK Unlink a module. F$UNLK

ASSEMBLER CALL: OS9 F$UNLK

MACHINE CODE: 103F 02

INPUT: (U) = Address of the module header.

OUTPUT: Nothing.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Tells OS-9 that the module is no longer needed by the calling process. The module's link count is decremented, and the module is destroyed and its memory deallocated when the link count equals zero. The module will not be destroyed if in use by any other process(es) because its link count will be non-zero. In Level Two systems, the module is usually switched out of the process' address space.

Device driver modules in use or certain system modules cannot be unlinked. ROMed modules can be unlinked but cannot be deleted from the module directory.

WAIT    Wait for child to die.    F$WAIT

ASSEMBLER CALL: OS9 F$WAIT

MACHINE CODE: 103F 04

INPUT: Nothing.

OUTPUT: (A) = Deceased child's process ID.
(B) = Child's exit status code.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

The calling process is deactivated until a child process terminates by executing an EXIT system call, or by receiving a signal. The child's ID number and exit status is returned to the parent. If the child died due to a signal, the exit status byte (B register) is the signal code.

If the caller has several children, the caller is activated when the first one dies, so one WAIT system call is required to detect termination of each child.

If a child died before the WAIT call, the caller is reactivated almost immediately. WAIT will return an error if the caller has no children.

See the EXIT description for more related information.

A64      Allocate a 64 byte memory block      F$A64
+++++

ASSEMBLER CALL:   OS9   F$A64

MACHINE CODE:   103F 30

INPUT: (X) = Base address of page table (zero if the page table has not yet been allocated).

OUTPUT: (A) = Block number.
(X) = Base address of page table.
(Y) = Address of block.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request is used to dynamically allocate 64 byte blocks of memory by splitting whole pages (256 byte) into four sections. The first 64 bytes of the base page are used as a "page table", which contains the MSB of all pages in the memory structure. Passing a value of zero in the X register will cause the F$A64 service request to allocate a new base page and the first 64 byte memory block. Whenever a new page is needed, an F$SRQM service request will automatically be executed. The first byte of each block contains the block number; routines using this service request should not alter it. Below is a diagram to show how 7 blocks might be allocated:

```
                     ANY 256 BYTE             ANY 256 BYTE
                     MEMORY PAGE              MEMORY PAGE

BASE PAGE --->     +---------------+        +---------------+
                   !               !        !X              !
                   !  PAGE TABLE   !        !   BLOCK 4     !
                   !  (64 bytes)   !        !  (64 bytes)   !
                   +---------------+        +---------------+
                   !X              !        !X              !
                   !   BLOCK 1     !        !   BLOCK 5     !
                   !  (64 bytes)   !        !  (64 bytes)   !
                   +---------------+        +---------------+
                   !X              !        !X              !
                   !   BLOCK 2     !        !   BLOCK 6     !
                   !  (64 byte)    !        !  (64 byte)    !
                   +---------------+        +---------------+
                   !X              !        !X              !
                   !   BLOCK 3     !        !   BLOCK 7     !
                   !  (64 byte)    !        !  (64 byte)    !
                   +---------------+        +---------------+
```

Block numbers range from 1..N

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

APRC Insert process in active process queue F$APRC
++++++

ASSEMBLER CALL: OS9 F$APRC

MACHINE CODE: 103F 2C

INPUT: (X) = Address of process descriptor.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request inserts a process into the active process queue so that it may be scheduled for execution.

All processes already in the active process queue are aged, and the age of the specified process is set to its priority. If the process is in system state, it is inserted after any other processes also in system state, but before any process in user state. If the process is in user state, it is inserted according to its age.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

FIND-64 Find a 64 byte memory block F$F64
+++++

ASSEMBLER CALL: OS9 F$F64

MACHINE CODE: 103F 2F

INPUT: (X) = Address of base page.
(A) = Block number.

OUTPUT: (Y) = Address of block.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request will return the address of a 64 byte memory block as described in the F$A64 service request. OS-9 used this service request to find process descriptiors and path descriptors when given their number.

Block numbers range from 1..N

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

IODEL Delete I/O device from system F$IODL

ASSEMBLER CALL: OS9 F$IODL

MACHINE CODE: 103F 33

INPUT: (X) = Address of an I/O module. (see description)

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request is used to determine whether or not an I/O module is being used. The X register passes the address of a device descriptor module, device driver module, or file manager module. The address is used to search the device table, and if found the use count is checked to see if it is zero. If it is not zero, an error condition is returned.

This service request is used primarily by IOMAN and may be of limited or no use for other applications.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

IOQUEUE Enter I/O queue F$IOQU

ASSEMBLER CALL: OS9 F$IOQU

MACHINE CODE: 103F 2B

INPUT: (A) = Process Number.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request links the calling process into the I/O queue of the specified process and performs an un-timed sleep. It is assumed that routines associated with the specified process will send a wakeup signal to the calling process.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

SETIRQ Add or remove device from IRQ table. F$IRQ
+++++

ASSEMBLER CALL: OS9 F$IRQ

MACHINE CODE: 103F 2A

INPUT: (X) = Zero to remove device from table, or the address of a packet as defined below to add a device to the IRQ polling table:

[x] = flip byte
[X+1] = mask byte
[X+2] = priority

(U) = Address of service routine's static storage area.
(Y) = Device IRQ service routine address.
(D) = Address of the device status register.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This service request is used to add a device to or remove a device from the IRQ polling table. To remove a device from the table the input should be (X)=Ø, (U)= Addr of service routine's static storage. This service request is primarily used by device driver routines. See the text of this manual for a complete discussion of the interrupt polling system.

PACKET DEFINITIONS:

Flip Byte — This byte selects whether the bits in the device status register are active when set or active when cleared. A set bit(s) identifies the active bit(s).

Mask Byte — This byte selects one or more bits within the device status register that are interrupt request flag(s). A set bit identifies an active bit(s).

Priority — The device priority number:
Ø = lowest
255 = highest

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

NXTPRCS Start next process F$NPRC

ASSEMBLER CALL: OS9 F$NPRC

MACHINE CODE: 103F 2D

INPUT: None.

OUTPUT: Control does not return to caller.

This system mode service request takes the next process out of the Active Process Queue and initiaites its execution. If there is no process in the queue, OS-9 waits for an interrupt, and then checks the active process queue again.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

R64 Deallocate a 64 byte memory block F$R64
+++++

ASSEMBLER CALL: OS9 F$R64

MACHINE CODE: 103F 31

INPUT: (X) = Address of the base page.
(A) = Block number.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request deallocates a 64 byte block of memory as described in the F$A64 service request.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

SRQMEM System memory request F$SRQM

ASSEMBLER CALL: OS9 F$SRQM

MACHINE CODE: 103F 28

INPUT: (D) = Byte count.

OUTPUT: (U) = Beginning address of memory area.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request allocates a block of memory from the top of available RAM of the specified size. The size requested is rounded to the next 256 byte page boundary.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

SRTMEM Return System Memory F$SRTM

ASSEMBLER CALL: OS9 F$SRTM

MACHINE CODE: 103F 29

INPUT: (U) = Beginning address of memory to return.
(D) = Number of bytes to return.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request is used to deallocate a block of contiguous 256 byte pages. The U register must point to an even page boundary.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

VMOD Verify module F$VMOD

ASSEMBLER CALL: OS9 F$VMOD

MACHINE CODE: 103F 2E

INPUT: (X) = Address of module to verify.

OUTPUT: (U) = Address of module directory entry.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system mode service request checks the moule header parity and CRC bytes of an OS-9 module. If these values are valid, then the module directory is searched for a module with the same name. If a module with the same name exists, the one with the highest revision level is retained in the module directory. Ties are broken in favor of the established module.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

ATTACH Attach a new device to the system. I$ATCH

ASSEMBLER CALL: OS9 I$ATCH

MACHINE CODE: 103F 80

INPUT: (X) = Address of device name string.
(A) = Access mode.

OUTPUT: (U) = Address of device table entry.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This service request is used to attach a new device to the system, or verify that it is already attached. The device's name string is used to search the system module directory to see if a device descriptor module with the same name is in memory (this is the name the device will be known by). The descriptor module will contain the name of the device's file manager, device driver and other related information. If it is found and the device is not already attached, OS-9 will link to its file manager and device driver, and then place their address' in a new device table entry. Any permanent storage needed by the device driver is allocated, and the driver's initialization routine is called (which usually initializes the hardware).

If the device has already been attached, it will not be reinitialized.

An ATTACH system call is not required to perform routine I/O. It does NOT "reserve" the device in question - it just prepares it for subsequent use by any process. Most devices are automatically installed, so it is used mostly when devices are dynamically installed or to verify the existance of a device.

The access mode parameter specifies which subsequent read and/or write operations will be permitted as follows:

- 0 = Use device capabilities.
- 1 = Read only.
- 2 = Write only.
- 3 = Both read and write.

CHDIR Change working directory. I$CDIR

ASSEMBLER CALL: OS9 I$CDIR

MACHINE CODE: 103F 86

INPUT: (X) = Address of the pathlist.
(A) = Access mode.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Changes a process' working directory to another directory file specified by the pathlist. The file must be a directory, and have read permission (public read if not owned by the calling process). New files may be added to the directory only if it has similar write permit attributes.

ACCESS MODES:

1 = Read
2 = Write
3 = Update (read or write)
4 = Execute

If the access mode is read, write, or update the current data directory is changed. If the access mode is execute, the current execution directory is changed.

CLOSE        Close a path to a file/device.        I$CLOS

ASSEMBLER CALL: OS9 I$CLOS

MACHINE CODE: 103F 8F

INPUT: (A) = Path number.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Terminates the I/O path specified by the path number. I/O can no longer be performed to the file/device, unless another OPEN or CREATE call is used. Devices that are non-sharable become available to other requesting processes. All OS-9 internally managed buffers and descriptors are deallocated.

Note: Because the OS9 F$EXIT service request automatically closes all open paths (except the standard I/O paths), it may not be necessary to close them individually with the OS9 I$CLOS service request.

Standard I/O paths are not typically closed except when it is desired to change the files/devices they correspond to.

CREATE    Create a path to a new file.    I$CREA

ASSEMBLER CALL: OS9 I$CREA

MACHINE CODE: 103F 83

INPUT: (X) = Address of the pathlist.
(A) = Access mode.
(B) = File attributes.

OUTPUT: (X) = Updated past the pathlist (trailing blanks skipped)
(A) = Path number.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Used to create a new file on a multifile mass storage device. The pathlist is parsed, and the new file name is entered in the specified (or default working) directory. The file is given the attributes passed in the B register, which has individual bits defined as follows:

bit Ø = read permit
bit 1 = write permit
bit 2 = execute permit
bit 3 = public read permit
bit 4 = public write permit
bit 5 = public execute permit
bit 6 = sharable file

The access mode parameter passed in register A must be either "WRITE" or "UPDATE". This only affects the file until it is closed; it can be reopened later in any access mode allowed by the file attributes (see OPEN). Files open for "WRITE" may allow faster data transfer than "UPDATE", which sometimes needs to pre-read sectors. These access codes are defined as given below:

2 = Write only.
3 = Update (read and write).

NOTE: If the execute bit (bit 2) is set, the file will be created in the working execution directory instead of the working data directory.

(continued)

CREATE (continued)

The path number returned by OS-9 is used to indentify the file in subsequent I/O service requests until the file is closed.

The file's owner is the user who created it. Other users may access the file only if the appropriate permission bit(s) are set in the file attributes byte.

No data storage is initially allocated; this is done automatically by WRITE and PUTSTAT calls.

An error will occur if the file name already exists in the directory. CREATE calls that specify non-multiple file devices (such as printers, terminals, etc.) work correctly: the CREATE behaves the same as OPEN. Create cannot be used to make directory files (see MAKDIR).

DELETE Delete a file. I$DLET

ASSEMBLER CALL: OS9 I$DLET

MACHINE CODE: 103F 87

INPUT: (X) = Address of pathlist.

OUTPUT: (X) = Updated past pathlist (trailing spaces skipped).

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This service request deletes the file specified by the pathlist. The file must have write permission attributes (public write if not the owner), and reside on a multifile mass storage device. Attempts to delete devices will result in an error.

DETACH Remove a device from the system. I$DTCH

ASSEMBLER CALL: OS9 I$DTCH

MACHINE CODE: 103F 81

INPUT: (U) = Address of the device table entry.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Removes a device from the system device table if not in use by any other process. The device driver's termination routine is called, then any permanent storage assigned to the driver is deallocated. The device driver and file manager modules associated with the device are unlinked (and may be destroyed if not in use by another process.

The I$DTCH service request must be used to un-attach devices that were attached with the I$ATCH service request. Both of these are used mainly by IOMAN and are of limited (cor no use) to the user. SCFMAN also uses ATTACH/DETACH to setup its second (echo) device.

DUP     Duplicate a path.     I$DUP

ASSEMBLER CALL: OS9 I$DUP

MACHINE CODE: 103F 82

INPUT: (A) = Path number of path to duplicate.

OUTPUT: (B) = New path number.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

Given the number of an existing path, returns another synonymous path number for the same file or device. SHELL uses this service request when it redirects I/O. Service requests using either the old or new path numbers operate on the same file or device.

NOTE: This only increments the "use count" of a path descriptor and returns the synonymous path number. The path descriptor is not copied.

GETSTAT          Get file/device status.          I$GSTT

ASSEMBLER CALL:    OS9 I$GSTT

MACHINE CODE:    103F 8D

INPUT: (A) = Path number.
(B) = Status code.
(Other registers depend upon status code)

OUTPUT: (depends upon status code)

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call is a "wild card" call used to handle individual device parameters that:

a) are not uniform on all devices
b) are highly hardware dependent
c) need to be user-changable

The exact operation of this call depends on the device driver and file manager associated with the path. A typical use is to determine a terminal's paramaters for backspace character, delete character, echo on/off, null padding, paging, etc. It is commonly used in conjunction with the SETSTAT service request which is used to set the device operating parameters. Below are the presently defined function codes for GETSTAT:

| NMEMONIC | CODE | FUNCTION |
|---|---|---|
| SS.OPT | 0 | Read the 32 byte option section of the path descriptor. |
| SS.RDY | 1 | Test for data ready on SCFMAN-type device. |
| SS.SIZ | 2 | Return current file size (on RBFMAN-type devices). |
| SS.POS | 5 | Get current file position. |
| SS.EOF | 6 | Test for end of file. |

(continued)

CODE
7-127 Reserved for future use.

CODE
128-255 These getstat codes and their parameter passing conventions are user definable (see the sections of this manual on writing device drivers). The function code and register stack are passed to the device driver.

Parameter Passing Conventions

The parameter passing conventions for each of these function codes are given below:

================================================================

SS.OPT (code 0): Read option section of the path descriptor.

INPUT: (A) = Path number
(B) = Function code 0
(X) = Address of place to put a 32 byte status packet.

OUTPUT: Status packet.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This getstat function reads the option section of the path descriptor and copies it into the 32 byte area pointed to by the X register. It is typically used to determine the current settings for echo, auto line feed, etc. For a complete description or the status packet, please see the section of this manual on path descriptors.

================================================================

GETSTAT (continued)

---

SS.RDY (code 1): Test for data available on SCFMAN supported devices.

INPUT: (A) = Path number.
(B) = Function code 1

OUTPUT:

| | Ready | Not Ready | Error |
|---|---|---|---|
| (CC) | C bit clear | C bit set | C bit set |
| (B) | zero | $F6 (E$NRDY) | ERROR Code |

---

SS.SIZ (code 2): Get current file size (RBFMAN supported devices only)

INPUT: (A) = Path number.
(B) = Function code 2

OUTPUT: (X) = M.S. 16 bits of current file size.
(U) = L.S. 16 bits of current file size.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

---

SS.POS (code 5): Get current file position (RBFMAN supported devices only).

INPUT: (A) = Path number
(B) = Function code 5

OUTPUT: (X) = M.S. 16 bits of current file position.
(U) = L.S. 16 bits of current file position.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

---

GETSTAT (continued)

---

SS.EOF (code 6): Test for end of file.

INPUT: (A) = Path number.
(B) = Function code 6

OUTPUT:

| | Not-EOF | EOF | ERROR |
|---|---|---|---|
| (CC) | C bit Clear | C bit set | C bit set |
| (B) | Zero | $D3 (E$EOF) | Error Code |

---

MAKDIR Make a new directory. I$MDIR

ASSEMBLER CALL: OS9 I$MDIR

MACHINE CODE: 103F 85

INPUT: (X) = Address of pathlist.
(B) = Directory attributes.

OUTPUT: (X) = Updated past pathlist (trailing spaces skipped).

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

MAKDIR is the only way a new directory file can be created. It will create and initialize a new directory as specified by the pathlist. The new directory file contains no entries, except for an entry for itself (".") and its parent directory ("..")

The caller is made the owner of the directory. MAKDIR does not return a path number because directory files are not "opened" by this request (use OPEN to do so). The new directory will automatically have its "directory" bit set in the access permission attributes. The remaining attributes are specified by the byte passed in the B register, which has individual bits defined as follows:

bit Ø = read permit
bit 1 = write permit
bit 2 = execute permit
bit 3 = public read permit
bit 4 = public write permit
bit 5 = public execute permit
bit 6 = sharable directory
bit 7 = (don't care)

OPEN | Open a path to a file or device. | I$OPEN

```
ASSEMBLER CALL:    OS9  I$OPEN

MACHINE CODE:      103F 84

INPUT:  (X) = Address of pathlist.
        (A) = Access mode (D S PE PW PR E W R)

OUTPUT: (X) = Updated past pathlist (trailing spaces skipped).
        (A) = Path number.

ERROR OUTPUT:    (CC) = C bit set.
                 (B)  = Appropriate error code.
```

Opens a path to an existing file or device as specified by the pathlist. A path number is returned which is used in subsequent service requests to identify the file.

The access mode parameter specifies which subsequent read and/or write operations are permitted as follows:

1 = read mode
2 = write mode
3 = update mode (both read and write)

Update mode can be slightly slower because sector pre-reads may be required for random access of bytes. The access mode must conform to the access permision attributes associated with the file or device (see CREATE). Only the owner may access a file unless the appropriate "public permit" bits are set.

Files can be opened by several processes (users) simultaneously. Devices have an attribute that specifies whether or not they are sharable on an individual basis.

NOTES: If the execution bit is set in the access mode, OS-9 will begin searching for the file in the working execution directory (unless the pathlist begins with a slash).

The sharable bit (bit 6) in the access mode can not lock other users out of a file in OS-9 Level I. It is present only for upward compatability with OS-9 Level II.

Directory files may be read or written if the D bit (bit 7) is set in the acces mode.

READ    Read data from a file or device.    I$READ

```
ASSEMBLER CALL:     OS9  I$READ

MACHINE CODE:       103F 89

INPUT:  (X) = Address to store data.
        (Y) = Number of bytes to read.
        (A) = Path number.

OUTPUT: (Y) = Number of bytes actually read.

ERROR OUTPUT:    (CC) = C bit set.
                 (B)  = Appropriate error code.
```

Reads a specified number of bytes from the path number given. The path must previously have been opened in READ or UPDATE mode. The data is returned exactly as read from the file/device without additional processing or editing such as backspace, line delete, end-of-file, etc.

AFTER all data in a file has been read, the next I$READ service request will return and end of file error.

NOTES: The keyboard abort, keyboard interrupt, and end-of-file characters may be filtered out of the input data on SCFMAN-type devices unless the corresponding entries in the path descriptor have been set to zero. It may be desirable to modify the device descriptor so that these values in the path descriptor are initialized to zero when the path is opened.

The number of bytes requested will be read unless:

A. An end-of-file occurs
B. An end-of-record occurs (SCFMAN only)
C. An error condition occurs.

READLN　　Read a text line with editing.　　I$RDLN

ASSEMBLER CALL: OS9 I$RDLN

MACHINE CODE: 103F 8B

INPUT: (X) = Address to store data.
(Y) = Maximum number of bytes to read.
(A) = Path number.

OUTPUT: (Y) = Actual number of bytes read.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call is the same as "READ" except it reads data from the input file or device until a carriage return character is encountered or until the maximum byte count specified is reached.

Line editing will occur on SCFMAN-type devices. Line editing refers to backspace, line delete, echo, automatic line feed, etc.

SCFMAN requires that the last byte entered be an end-of-record character (normally carriage return). If more data is entered that the maximum specified, it will not be accepted and a PD.OVF character (normally bell) will be echoed.

After all data in a file has been read, the next I$RDLN service request will return an end of file error.

NOTE: For more information on line editing, see the section of this manual on "SCFMAN line editing features".

SEEK Reposition the logical file pointer. I$SEEK

ASSEMBLER CALL: OS9 I$SEEK

MACHINE CODE: 103F 88

INPUT: (A) = Path number.
(X) = M.S. 16 bits of desired file position.
(U) = L.S. 16 bits of desired file position.

OUTPUT: None.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call repositions the "file pointer"; the 32-bit address of the the next byte in the file to be read from or written to.

A seek may be performed to any value even if the file is not large enough. Subsequent WRITEs will automatically expand the file to the required size (if possible), but READs will return an end-of-file condition. Note that a SEEK to address zero is the same as a "rewind" operation.

Seeks to non-random access devices are usually ignored and return without error.

SETSTAT Set file/device status. I$SSTT

ASSEMBLER CALL: OS9 I$SSTT

MACHINE CODE: 103F 8E

INPUT: (A) = Path number.
(B) = Function code.
(Other registers depend upon the function code).

OUTPUT: (Depends upon the function code).

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call is a "wild card" call used to handle individual device parameters that:

a) are not uniform on all devices
b) are highly hardware dependant
c) need to be user-chagable

The exact operation of this call depends on the device driver and file manager associated with the path. A typical use is to set a terminal's parameters for backspace character, delete character, echo on/off, null padding, paging etc. It is commonly used in conjuction with the GETSTAT service request which is used to read the device's operating parameters etc. Below are the presently defined function codes:

| NMEMONIC | CODE | FUNCTION |
|---|---|---|
| SS.OPT | $0 | Write the 32 byte option section of the path descriptor to set the device |
| SS.SIZ | $2 | Set the file size (for RBFMAN type devices only). |
| SS.RST | $3 | Restore head to track zero. |
| SS.WRT | $4 | Write track. |

SETSTAT (continued)

CODES
5-127 Reserved for future use.

CODES
128-255 These SETSTAT codes and their parameter passing conventions are user definable (see the sections of this manual on writing device drivers). The function code and register stack are passed to the device driver.

Parameter Passing Conventions

The parameter passing conventions for each of these function codes is given below:

==========================================================

SS.OPT (code Ø): Write option section of path descriptor.

```
INPUT:  (A) = Path number
        (B) = Function code Ø
        (X) = Address of a 32 byte status packet

OUTPUT: None.

ERROR OUTPUT:  (CC) = C bit set
               (B)  = Appropriate error code.
```

This setstat function writes the option section of the path descriptor from the 32 byte status packet pointed to by the X register. It is typically used to set the device operating parameters, such as echo, auto line feed, etc. For a complete description of what is in the status packet, please see the section of this manual on path descriptors.

==========================================================

SETSTAT (continued)

===========================================================

SS.SIZ (code 2): Set file size (RBFMAN-type devices)

```
INPUT:  (A) = Path number
        (B) = Function code 2
        (X) = M.S. 16 bits of desired file size.
        (U) = L.S. 16 bits of desired file size.

OUTPUT: None.

ERROR OUTPUT:  (CC) = C bit set
               (B)  = Appropriate error code.
```

This setstat function is used to set the file size (RBFMAN supported devices only).

===========================================================

SS.RST (code 3): Restore head to track zero.

```
INPUT:  (A) = Path number
        (B) = Function code 3

OUTPUT: None

ERROR OUTPUT:  (CC) = C bit set
               (B)  = Appropriate error code
```

===========================================================

SS.WTK (code 4): Write track.

```
INPUT:  (A) = Path number
        (B) = Function code 4
        (X) = Address of track buffer.
        (U) = Track number (L.S. 8 bits)
        (Y) = Side/density

              Bit B0 = SIDE    (0 = side zero, 1 = side one)

              Bit B1 = DENSITY (0 = single, 1 = double)

OUTPUT: None

ERROR OUTPUT:  (CC) = C bit set
               (B)  = Appropriate error code
```

===========================================================

WRITE Write data to a file or device. I$WRIT

ASSEMBLER CALL: OS9 I$WRIT

MACHINE CODE: 103F 8A

INPUT: (X) = Address of data to write.
(Y) = Number of bytes to write.
(A) = Path number.

OUTPUT: (Y) = Number of bytes actually written.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

WRITE outputs one or more bytes to a file or device associated with the path number specified. The path must have been OPENed or CREATEed in the WRITE or UPDATE access modes.

Data is written to the file or device without processing or editing. If data is written past the present end-of-file, the file is automatically expanded.

WRITELN    Write a line of text with editing.    I$WRLN

ASSEMBLER CALL: OS9 I$WRLN

MACHINE CODE: 103F 8C

INPUT: (X) = Address of data to write.
(Y) = Maximum number of bytes to write.
(A) = Path number.

OUTPUT: (Y) = Actual number of bytes written.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.

This system call is similar to WRITE except it writes data until a carriage return character is encountered. Line editing is also activated for character-oriented devices such as terminals, printers, etc. The line editing refers to auto line feed, null padding at end-of-line, etc.

For more information about line editing, see the section of this manual on "SCFMAN Line Editing Features".

# WRITING RBF-TYPE DEVICE DRIVERS

An RBF type device driver module contains a package of subroutines that perform sector oriented I/O to or from a specific hardware controller. These modules are usually reentrant so that one copy of the module can simultaneously run several different devices that use identical I/O controllers. IOMAN will allocate a static storage area for each device (which may control several drives). The size of the storage area is given in the device driver module header. Some of this storage area will be used by IOMAN and RBFMAN, the device driver is free to use the remainder in any way that it desires. This static storage is laid out as follows:

## Static Storage Definitions

```
    OFFSET               ORG Ø

      Ø        V.PAGE    RMB 1    PORT EXTENDED ADDRESS
      1        V.PORT    RMB 2    DEVICE BASE ADDRESS
      3        V.LPRC    RMB 1    LAST ACTIVE PROCESS ID
      4        V.BUSY    RMB 1    ACTIVE PROCESS ID (Ø = NOT BUSY)
      5        V.WAKE    RMB 1    PROCESS ID TO REAWAKEN
               V.USER    EQU .    END OF OS9 DEFINITIONS

      6        V.NDRV    RMB 1    NUMBER OF DRIVES
               DRVBEG    EQU .    BEGINNING OF DRIVE TABLES
      7        TABLES    RMB DRVMEM*N  RESERVE N DRIVE TABLES
               FREE      EQU .    FREE FOR DRIVER TO USE
```

NOTE: V.PAGE through V.USER are predefined in the OS9DEFS file.
V.NDRV, DRVBEG, DRVMEM are predefined in the RBFDEFS file.

V.PAGE, V.PORT — These three bytes are defined by IOMAN to be the 24 bit device address.

V.LPRC — This location contains the process-ID of the last process to use the device. Not used by RBF-type device drivers.

V.BUSY — This location contains the process-ID of the process currently using the device. Defined by RBFMAN.

V.WAKE — This location contains the process-ID of any process that is waiting for the device to complete I/O (Ø = NO PROCESS WAITING). Defined by device driver.

V.NDRV — This location contains the number of drives that the controller will be working with. Defined by the device driver as the maximum number of drives that the controller can work with. RBFMAN will assume that there is a drive table for each drive. Also see the driver INIT routine in this section.

TABLES — This area contains one table for each drive that the controller will handle (RBFMAN will assume that there are as many tables as indicated by V.NDRV). Some time after the driver INIT routine has been called, RBFMAN will issue a request for the driver to read the identification sector (logical sector zero) from a drive. At this time, the driver will initialize the corresponding drive table by copying the first part of the identification sector (up to DD.SIZ) into it. Also see the "Identification Sector" section of this manual. The format of each drive table is as given below:

```
OFFSET          ORG Ø

$ØØ    DD.TOT   RMB 3   TOTAL NUMBER OF SECTORS
$Ø3    DD.TKS   RMB 1   TRACK SIZE ( IN SECTORS )
$Ø4    DD.MAP   RMB 2   # BYTES IN ALLOCATION BIT MAP
$Ø6    DD.BIT   RMB 2   NUMBER OF SECTORS / BIT
$Ø8    DD.DIR   RMB 3   ADDRESS OF ROOT DIRECTORY
$ØB    DD.OWN   RMB 2   OWNER'S USER NUMBER
$ØD    DD.ATT   RMB 1   DISK ATTRIBUTES
$ØE    DD.DSK   RMB 2   DISK ID
$1Ø    DD.FMT   RMB 1   MEDIA FORMAT
$11    DD.SPT   RMB 2   SECTORS/TRACK
$15    DD.RES   RMB 2   RESERVED FOR FUTURE USE
       DD.SIZ   EQU .

$15    V.TRAK   RMB 2   CURRENT TRACK NUMBER
$17    V.BMB    RMB 1   BIT-MAP USE FLAG
$18    DRVMEM   EQU .   SIZE OF EACH DRIVE TABLE
```

DD.TOT — This location contains the total number of sectors contained on the disk.

DD.TKS — This location contains the track size (in sectors).

DD.MAP — This location contains the number of bytes in the disk allocation bit map.

DD.BIT This location contains the number of sectors that each bit represents in the disk allocation bit map.

DD.DIR This location contains the logical sector number of the disk root directory.

DD.OWN This location contains the disk owner's user number.

DD.ATT This location contains the disk access permission attributes as defined below:

```
BIT 7 = D       (DIRECTORY IF SET)
BIT 6 = S       (SHARABLE IF SET)
BIT 5 = PX      (PUBLIC EXECUTE IF SET)
BIT 4 = PW      (PUBLIC WRITE IF SET)
BIT 3 = PR      (PUBLIC READ IF SET)
BIT 2 = X       (EXECUTE IF SET)
BIT 1 = W       (WRITE IF SET)
BIT Ø = R       (READ IF SET)
```

DD.DSK This location contains a pseudo random number which is used to identify a disk so that OS-9 may detect when a disk is removed from the drive and another inserted in its place.

DD.FMT DISK FORMAT:

```
BIT  BØ - SIDE
          Ø = SINGLE SIDED
          1 = DOUBLE SIDED

BIT  B1 - DENSITY
          Ø = SINGLE DENSITY
          1 = DOUBLE DENSITY

BIT  B2 - TRACK DENSITY
          Ø = SINGLE (48 TPI)
          1 = DOUBLE (96 TPI)
```

DD.SPT Number of sectors per track (track zero may use a different value, specified by IT.TØS in the device descriptor).

DD.RES RESERVED FOR FUTURE USE

V.TRAK This location contains the current track which the head is on and is updated by the driver.

V.BMB This location is used by RBFMAN to indicate whether or not the disk allocation bit map is currently in use (Ø = not in use). The disk driver routines must not alter this location.

## Other Important Parameters

Other parameters which may be important to device drivers can found in the path descriptor. For a complete description of the values which are contained in the path descriptor, please see the section of this manual on "RBFMAN Definitions of The Path Descriptor". Also see the section on device descriptors (especially the initialization table).

## RBFMAN Device Driver Subroutines

As with all device drivers, RBFMAN-type device drivers use a standard executable memory module format with a module type of "device driver" (CODE $E). The execution offset address in the module header points to a branch table that has six three byte entries. Each entry is typically a LBRA to the corresponding subroutine. The branch table is defined as follows:

```
ENTRY   LBRA    INIT     INITIALIZE DRIVE
        LBRA    READ     READ SECTOR
        LBRA    WRITE    WRITE SECTOR
        LBRA    GETSTA   GET STATUS
        LBRA    SETSTA   SET STATUS
        LBRA    TERM     TERMINATE DEVICE
```

Each subroutine should exit with the condition code register C bit cleared if no error occured. Otherwise the C bit should be set and an appropriate error code returned in the B register. Below is a description of each subroutine, its input parameters, and its output parameters.

NAME: INIT

INPUT: (U) = ADDRESS OF DEVICE STATIC STORAGE
(Y) = ADDRESS OF THE DEVICE DESCRIPTOR MODULE

OUTPUT: NONE

ERROR OUTPUT: (CC) = C BIT SET
(B) = ERROR CODE

FUNCTION: INITIALIZE DEVICE AND ITS STATIC STORAGE AREA

1. If disk writes are verified, use the F$SRQM service request to allocate a 256 byte buffer area where a sector may be read back and verified after a write.

2. Initialize the device permanent storage. For floppy disk controller typically this consists of initializing V.NDRV to the number of drives that the controller will work with, initializing DD.TOT in the drive table to a non-zero value so that sector zero may be read or written to, and initializing V.TRAK to $FF so that the first seek will find track zero.

3. Place the IRQ service routine on the IRQ polling list by using the OS9 F$IRQ service request.

4. Initialize the device control registers (enable interrupts if necessary).

NOTE: Prior to being called, the device permanent storage will be cleared (set to zero) except for V.PAGE and V.PORT which will contain the 24 bit device address. The driver should initialize each drive table appropriately for the type of disk the driver "expects" on the corresponding drive.

NAME: READ

INPUT: (U) = ADDRESS OF THE DEVICE STATIC STORAGE
(Y) = ADDRESS OF THE PATH DESCRIPTOR
(B) = MSB OF DISK LOGICAL SECTOR NUMBER
(X) = LSB's OF DISK LOGICAL SECTOR NUMBER

OUTPUT: SECTOR IS RETURNED IN THE SECTOR BUFFER

ERROR OUTPUT: (CC) = C BIT SET
(B) = APPROPRIATE ERROR CODE

FUNCTION: READ A 256 BYTE SECTOR

Read a sector from the disk and place it in the sector buffer (256 byte). Below are the things that the disk driver must do:

1. Get the sector buffer address from PD.BUF in the path descriptor.

2. Get the drive number from PD.DRV in the path descriptor.

3. Compute the physical disk address from the logical sector number.

4. Initiate I/O.

5. Move V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete (the IRQ service routine is responsible for sending a wake up signal). After awakening, test V.WAKE to see if it is clear, if not, go back to sleep.

If the disk controller can not be interrupt driven it will be necessary to perform programmed I/O.

NOTE1: Whenever logical sector zero is read, the first part of this sector must be copied into the proper drive table (get the drive number from PD.DRV in the path descriptor). The number of bytes to copy is DD.SIZ.

NOTE2: The drive number (PD.DRV) should be used to compute the offset to the corresponding drive table as follows:

```
LDA PD.DRV,Y   Get drive number
LDB #DRVMEM    Get size of a drive table
MUL
LEAX DRVBEG,U  Get address of first table
LEAX D,X       Compute address of table N
```

NAME: WRITE

INPUT: (U) = ADDRESS OF THE DEVICE STATIC STORAGE AREA
(Y) = ADDRESS OF THE PATH DESCRIPTOR
(B) = MSB OF THE DISK LOGICAL SECTOR NUMBER
(X) = LSB's OF THE DISK LOGICAL SECTOR NUMBER

OUTPUT: THE SECTOR BUFFER IS WRITTEN OUT TO DISK

ERROR OUTPUT: (CC) = C BIT SET
(B) = APPROPRIATE ERROR CODE

FUNCTION: WRITE A SECTOR

Write the sector buffer (256 byte) out to the disk. Below are the things that a disk driver must do:

1. Get the sector buffer address from PD.BUF in the path descriptor.

2. Get the drive number from PD.DRV in the path descriptor.

3. Compute the physical disk address from the logical sector number.

4. Initiate I/O.

5. Move V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete (the IRQ service routine is responsible for sending the wakeup signal). After awakening, test V.WAKE to see if it is clear, if it is not, then go back to sleep.

If the disk controller can not be interrupt driven, it will be necessary to perform programmed I/O.

6. If PD.VFY in the path descriptor is equal to zero, read the sector back in and verify that it was written correctly. This usually does not involve a compare of the data.

NOTE1: If disk writes are to be verified, the INIT routine must request the buffer where the sector may be placed when it is read back in. Do not copy sector zero into the drive table when it is read back to be verified.

NOTE3: Use the drive number (PD.DRV) to compute the offset to the corresponding drive table as shown for the READ routine.

NAME: GETSTA
PUTSTA

INPUT: (U) = ADDRESS OF THE DEVICE STATIC STORAGE AREA
(Y) = ADDRESS OF THE PATH DESCRIPTOR
(A) = STATUS CODE

OUTPUT: (DEPENDS UPON THE FUNCTION CODE)

ERROR OUTPUT: (CC) = C BIT SET
(B) = APPROPRIATE ERROR CODE

FUNCTION: GET / SET DEVICE STATUS

These routines are wild card calls used to get (set) the device's operating parameters as specified for the OS9 I$GSTT and I$SSTT service requests. The codes passed to the device driver are given below:

GETSTAT: Any I$GSTT function code not defined by Microware

SETSTAT: SS.RST (code 3) = Restore head to track zero.

SS.WRT (code 4) = Write track.

Any I$SSTT function code not defined by Microware.

It may be necessary to examine or change the register stack which contains the values of MPU registers at the time of the I$GSTT or I$SSTT service request. The address of the register stack may be found in PD.RGS, which is located in the path descriptor. The following offsets may be used to access any particular value in the register stack:

| OFFSET | NMEMONIC | | MPU REGISTER |
|---|---|---|---|
| $Ø | R$CC | RMB 1 | CONDITION CODE REGISTER |
| $1 | R$D | EQU . | D REGISTER |
| $1 | R$A | RMB 1 | A REGISTER |
| $2 | R$B | RMB 1 | B REGISTER |
| $3 | R$DP | RMB 1 | DP REGISTER |
| $4 | R$X | RMB 2 | X REGISTER |
| $6 | R$Y | RMB 2 | Y REGISTER |
| $8 | R$U | RMB 2 | U REGISTER |
| $A | R$PC | RMB 2 | PROGRAM COUNTER |

NAME: TERM

INPUT: (U) = ADDRESS OF DEVICE STATIC STORAGE AREA

OUTPUT: NONE

ERROR OUTPUT: (CC) = C BIT SET
(B) = APPROPRIATE ERROR CODE

FUNCTION: TERMINATE DEVICE

This routine is called when a device will no longer be needed in the system (when the link count goes to zero). The primary things that it does are:

1. Wait until any pending I/O has completed.

2. Disable the device interrupts.

3. Remove the device from the IRQ polling list.

4. If the INIT routine reserved a 256 byte buffer for verifying disk writes, return the memory with the F$MEM service request.

NAME: THE IRQ SERVICE ROUTINE

FUNCTION: SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device drivers branch table and not called directly from RBFMAN, it is an important routine in device drivers. The main things that it does are:

1. Service device interrupts.

2. When the I/O is complete, the IRQ service routine should send a wake up signal to the process whose process ID is in V.WAKE

Also clear V.WAKE as a flag to the mainline program that the IRQ has indeed occurred.

NOTE: When the IRQ service routine finishes servicing an interrupt it must clear the carry and exit with an RTS instruction.

## WRITING SCF-TYPE DEVICE DRIVERS

An SCFMAN-type device driver module contains a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. These modules are usually reentrant so that one copy of the module can simultaneously run several different devices that use identical I/O controllers. For each "incarnation" of the driver, IOMAN will allocate a static storage area for that device. The size of the storage area is given in the device driver module header. Some of this storage area will be used by IOMAN and SCFMAN, the device driver is free to use the remainder in any way it desires (typically as variables and buffers). This static storage is laid out as given below:

STATIC STORAGE DEFINITIONS

```
     OFFSET          ORG 0

      $0     V.PAGE  RMB 1    PORT EXTENDED ADDRESS
      $1     V.PORT  RMB 2    DEVICE BASE ADDRESS
      $3     V.LPRC  RMB 1    LAST ACTIVE PROCESS ID
      $4     V.BUSY  RMB 1    ACTIVE PROCESS ID (0 = NOT BUSY)
      $5     V.WAKE  RMB 1    PROCESS ID TO REAWAKEN
             V.USER  EQU .    END OF OS9 DEFINITIONS

      $6     V.TYPE  RMB 1    DEVICE TYPE OR PARITY
      $7     V.LINE  RMB 1    LINES LEFT TILL END OF PAGE
      $8     V.PAUS  RMB 1    PAUSE REQUEST (0 = NO PAUSE)
      $9     V.DEV2  RMB 2    ATTACHED DEVICE STATIC STORAGE
      $B     V.INTR  RMB 1    INTERRUPT CHARACTER
      $C     V.QUIT  RMB 1    QUIT CHARACTER
      $D     V.PCHR  RMB 1    PAUSE CHARACTER
      $E     V.ERR   RMB 1    ERROR ACCUMULATOR
      $F     V.SCF   EQU .    END OF SCFMAN DEFINITIONS

             FREE    EQU .    FREE FOR DEVICE DRIVER TO USE
```

V.PAGE, V.PORT — These three bytes are defined by IOMAN to be the 24 bit device address.

V.LPRC — This location contains the process-ID of the last process to use the device. The IRQ service routine is responsible for sending this process the proper signal in case a "QUIT" character or an "INTERRUPT" character is recieved. Defined by SCFMAN.

V.ERR — This location is used to accumulate I/O errors. Typically it is used by the IRQ service routine to record errors so that they may be reported later when SCFMAN calls one of the device driver routines.

## SCFMAN DEVICE DRIVER SUBROUTINES

As with all device drivers, SCFMAN device drivers use a standard executable memory module format with a module type of "device driver" (CODE $E). The execution offset address in the module header points to a branch table that has six three byte entries. Each entry is typically a LBRA to the corresponding subroutine. The branch table is as follows:

```
ENTRY   LBRA  INIT       INITIALIZE DEVICE
        LBRA  READ       READ CHARACTER
        LBRA  WRITE      WRITE CHARACTER
        LBRA  GETSTA     GET DEVICE STATUS
        LBRA  SETSTA     SET DEVICE STATUS
        LBRA  TERM       TERMINATE DEVICE
```

Each subroutine should exit with the condition code register C bit cleared if no error occured. Otherwise the C bit should be set and an appropriate error code returned in the B register. Below is a description of each subroutine, its input parameters and its output parameters.

NAME: INIT

INPUT: (U) = ADDRESS OF DEVICE STATIC STORAGE
(Y) = ADDRESS OF DEVICE DESCRIPTOR MODULE

OUTPUT: NONE

ERROR OUTPUT: (CC) = C BIT SET
(B) = ERROR CODE

FUNCTION: INITIALIZE DEVICE AND ITS STATIC STORAGE

1. Initialize the device static storage.

2. Place the IRQ service routine on the IRQ polling list by using the OS9 F$IRQ service request.

3. Initialize the device control registers (enable interrupts if necessary).

NOTE: Prior to being called, the device static storage will be cleared (set to zero) except for V.PAGE and V.PORT which will contain the 24 bit device address. There is no need to initialize the portion of static storage used by IOMAN and SCFMAN.

NAME: READ

INPUT: (U) = ADDRESS OF DEVICE STATIC STORAGE
(Y) = ADDRESS OF PATH DESCRIPTOR

OUTPUT: (A) = CHARACTER READ

ERROR OUTPUT: (CC) = C BIT SET
(B) = ERROR CODE

FUNCTION: GET NEXT CHARACTER

This routine should get the next character from the input buffer. If there is no data ready, this routine should copy its process ID from V.BUSY into V.WAKE and then use the F$SLEP service request to put itself to sleep.

Later when data is recieved, the IRQ service routine will leave the data in a buffer, then check V.WAKE to see if any process is waiting for the device to complete I/O. If so, the IRQ service routine should send a wakeup signal to it.

NOTE: Data buffers are NOT automatically allocated. If any are used, they should be defined somewhere in the device's static storage area.

NAME: WRITE

INPUT: (U) = ADDRESS OF DEVICE STATIC STORAGE
(Y) = ADDRESS OF THE PATH DESCRIPTOR
(A) = CHAR TO WRITE

OUTPUT: NONE

ERROR OUTPUT: (CC) = C BIT SET
(B) = ERROR CODE

FUNCTION: OUTPUT A CHARACTER

This routine places a data byte into an output buffer and enables the device output interrupts. If the data buffer is already full, this routine should copy its process ID from V.BUSY into V.WAKE and then put itself to sleep.

Later when the IRQ service routine transmits a character and makes room for more data in the buffer, it will check V.WAKE to see if there is a process waiting for the device to complete I/O. If there is, it will send a wake up signal to that process.

Note: This routine must ensure that the IRQ service routine will start up when data is placed into the buffer. After an interrupt is generated the IRQ service routine will continue to transmit data until the data buffer is empty, and then it will disable the device's "ready to transmit" interrupts.

Note: Data buffers are NOT automatically allocated. If any are used, they should be defined somewhere in the device's static storage.

NAME: GETSTA
SETSTA

INPUT: (U) = ADDRESS OF DEVICE STATIC STORAGE
(Y) = ADDRESS OF PATH DESCRIPTOR
(A) = STATUS CODE

OUTPUT: ( DEPENDS UPON FUNCTION CODE )

FUNCTION: GET / SET DEVICE STATUS

This routine is a wild card call used to get (set) the device parameters specified in the I$GSTT and I$SSTT service requests. Currently all of the function codes defined by Microware for SCF-type devices are handled by IOMAN or SCFMAN. Any codes not defined by microware will be passed to the device driver.

It may be necessary to examine or change the register packet which contains the values of the 6809 registers at the time the OS9 service request was issued. The address of the register packet may be found in PD.RGS, which is located in the path descriptor. The following offsets may be used to access any particular value in the register packet:

| OFFSET | NMEMONIC | | | MPU REGISTER |
|---|---|---|---|---|
| $0 | R$CC | RMB | 1 | CONDITIONS CODE REGISTER |
| $1 | R$D | EQU | . | D REGISTER |
| $1 | R$A | RMB | 1 | A REGISTER |
| $2 | R$B | RMB | 1 | B REGISTER |
| $3 | R$DP | RMB | 1 | DP REGISTER |
| $4 | R$X | RMB | 2 | X REGISTER |
| $6 | R$Y | RMB | 2 | Y REGISTER |
| $8 | R$U | RMB | 2 | U REGISTER |
| $A | R$PC | RMB | 2 | PROGRAM COUNTER |

NAME: TERM

INPUT: (U) = PTR TO DEVICE STATIC STORAGE

OUTPUT: NONE

ERROR OUTPUT: (CC) = C bit set
(B) = Appropriate error code

FUNCTION: TERMINATE DEVICE

This routine is called when a device will no longer be needed (when its link count goes to zero). The main things that it does are:

1. Wait until the output buffer has been emptied (by the IRQ service routine).

2. Disable device interrupts.

3. Remove device from the IRQ polling list.

NOTE: Static storage used by device drivers is never returned to the free memory pool. Therefore, it is desirable to NEVER terminate any device that might be used again. Modules contained in the BOOT file will NEVER be terminated.

NAME: IRQ SERVICE ROUTINE

FUNCTION: SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device drivers branch table and not called directly from SCFMAN, it is an important routine in device drivers. The main things that it does are:

1. Service the device interrupts (recieve data from device or send data to it). This routine should put its data into and get its data from buffers which you have defined in the device static storage.

2. Wake up any process waiting for I/O to complete by checking to see if there is a process ID in V.WAKE (non-zero) and if so send a wakeup signal to that process.

3. If the device is ready to send more data and the output buffer is empty, disable the device's "ready to transmit" interrupts.

4. If a pause character is recieved, set V.PAUS in the attached device static storage to a non-zero value. The address of the attached device static storage is in V.DEV2.

When the IRQ service routine finishes servicing an interrupt, it must clear the carry and exit with an RTS instruction.

# WRITING A SYSTEM BOOTSTRAP MODULE

The bootstrap module contains one subroutine that loads the bootstrap file and some related information into memory. It uses the standard executable module format with a module type of "system" (code $C). The execution offset in the module header contains the offset to the entry point of this subroutine. The following section describes the parameters passed to the the BOOT routine and its function. Also see the sections of this manual on "Writing RBF-type Device Drivers" and "Logical and Physical Disk Organization".

NAME: BOOT

INPUT: None.

OUTPUT: (D) = SIZE OF THE BOOT FILE (in bytes)
(X) = ADDRESS OF WHERE THE BOOT FILE WAS LOADED IN MEMORY

ERROR OUTPUT: (CC) = C BIT SET
(B) = APPROPRIATE ERROR CODE

FUNCTION: LOAD THE BOOT FILE INTO MEMORY FROM MASS-STORAGE

This routine attempts to load the bootstrap file into memory from a mass-storage device. Typically it will read some form of identification block which will contain the location and size of the bootstrap file. OS-9 is called to allocate a memory area large enough for the boot file, and then it loads the boot file into this memory area. Below is a description of how this is done for RBFMAN-type devices (DISK):

1. Read the identification sector (sector zero) from the disk. BOOT must pick its own buffer area.

The identification sector contains the values for DD.BT (the 24 bit logical sector number of the bootstrap file), and DD.BSZ (the size of the bootstrap file in bytes). For a full description of the identification sector, please see the section on "Physical and Logical Disk Organization".

2. After reading the identification sector into the buffer, get the 24 bit logical sector number of the bootstrap file from DD.BT.

3. Get the size (in bytes) of the bootstrap file from DD.BSZ. The boot is contained in one logically contiguous block beginning at the logical sector specified in DD.BT and extending for ( DD.BSZ/256 + 1 ) sectors.

4. Use the OS9 F$SRQM service request to request the memory area where the boot file will be loaded into.

5. Read the boot file into this memory area.

6. Return the size of the boot file and its location.

```
MODULE            EXECUTABLE MEMORY MODULE FORMAT
OFFSET

          +------------------------------+   ---+---------+---
  $00     !                              !      !         !
          +--     Sync Bytes ($87CD)   --+      !         !
  $01     !                              !      !         !
          +------------------------------+      !         !
  $02     !                              !      !         !
          +--    Module Size (bytes)   --+      !         !
  $03     !                              !      !         !
          +------------------------------+      !         !
  $04     !                              !      !         !
          +--    Module Name Offset    --+   header       !
  $05     !                              !   parity       !
          +------------------------------+      !         !
  $06     !     Type     !   Language    !      !         !
          +------------------------------+      !         !
  $07     !  Attributes  !   Revision    !      !         !
          +------------------------------+   ---+---    module
  $08     !    Header Parity Check       !                CRC
          +------------------------------+                 !
  $09     !                              !                 !
          +--     Execution Offset     --+                 !
  $0A     !                              !                 !
          +------------------------------+                 !
  $0B     !                              !                 !
          +--  Permanent Storage Size  --+                 !
  $0C     !                              !                 !
          +------------------------------+                 !
  $0D     !                              !                 !
          ! (Add'l optional header       !                 !
          !  extensions located here)    !                 !
          !                              !                 !
          !  .  .  .  .  .  .  .  .  .   !                 !
          !                              !                 !
          !                              !                 !
          !        Module Body           !                 !
          ! object code, constants, etc. !                 !
          !                              !                 !
          !                              !                 !
          +------------------------------+                 !
          !                              !                 !
          +--                          --+                 !
          !      CRC Check Value         !                 !
          +--                          --+                 !
          !                              !                 !
          +------------------------------+   ------------+---
```

```
MODULE     DEVICE DESCRIPTOR MODULE FORMAT
OFFSET

           +-------------------------------+  ---+--------+---
  $0       !                               !     !        !
           +--     Sync Bytes ($87CD)    --+     !        !
  $1       !                               !     !        !
           +-------------------------------+     !        !
  $2       !                               !     !        !
           +--     Module Size (bytes)     !     !        !
  $3       !                               !     !        !
           +-------------------------------+     !        !
  $4       !                               !     !        !
           +-- Offset to Module Name     --+   header     !
  $5       !                               !   parity     !
           +-------------------------------+     !        !
  $6       ! $F (TYPE)     !  $1 (LANG)    !     !        !
           +-------------------------------+     !        !
  $7       ! Atributes     !    Revision   !     !        !
           +-------------------------------+  ---+---     !
  $8       !  Header Parity Check          !              !
           +-------------------------------+              !
  $9       !                               !              !
           +--   Offset to File Manager  --+              !
  $A       !            Name String        !            module
           +-------------------------------+             CRC
  $B       !                               !              !
           +-- Offset to Device Driver   --+              !
  $C       !            Name String        !              !
           +-------------------------------+              !
  $D       !          Mode Byte            !              !
           +-------------------------------+              !
  $E       !                               !              !
           +--     Device Controller     --+              !
  $F       ! Absolute Physical Address     !              !
           +--        (24 bit)           --+              !
  $10      !                               !              !
           +-------------------------------+              !
  $11      !   Option Table Size           !              !
           +-------------------------------+              !
$12,$12+N  !                               !              !
           !    (Option Table)             !              !
           !                               !              !
           ! . . . . . . . . . . . . . .   !              !
           !                               !              !
           !    (Name Strings etc)         !              !
           !                               !              !
           +-------------------------------+              !
           !                               !              !
           +--                           --+              !
           !    CRC Check Value            !              !
           +--                           --+              !
           !                               !              !
           +-------------------------------+  ------------+---
```

```
MODULE      CONFIGURATION MODULE FORMAT
OFFSET

         +-------------------------------+   ---+--------+---
 $00     !                               !      !        !
         +--     Sync Bytes ($87CD)    --+      !        !
 $01     !                               !      !        !
         +-------------------------------+      !        !
 $02     !                               !      !        !
         +--    Module Size (bytes)    --+      !        !
 $03     !                               !      !        !
         +-------------------------------+      !        !
 $04     !                               !      !        !
         +--    Module Name Offset     --+    header     !
 $05     !                               !    parity     !
         +-------------------------------+      !        !
 $06     !  $C (TYPE)    !  0   (LANG)   !      !        !
         +-------------------------------+      !        !
 $07     !  Attributes   !   Revision    !      !        !
         +-------------------------------+   ---+---   module
 $08     !     Header Parity Check       !               CRC
         +-------------------------------+                !
 $09     !                               !                !
         +--    Forced Limit of Top    --+                !
 $0A     !        of Free RAM            !                !
         +--                           --+                !
 $0B     !                               !                !
         +-------------------------------+                !
 $0C     ! # IRQ Polling Table Entries   !                !
         +-------------------------------+                !
 $0D     !    # Device Table Entries     !                !
         +-------------------------------+                !
 $E      !                               !                !
         +--    Offset to Startup      --+                !
 $0F     !      Module Name Sring        !                !
         +-------------------------------+                !
 $10     !                               !                !
         +-- Offset to Default Mass-   --+                !
 $11     !  Storage Device Name String   !                !
         +-------------------------------+                !
 $12     !                               !                !
         +--    Offset to Bootstrap    --+                !
 $13     !      Module Name String       !                !
         +-------------------------------+                !
 $14     !                               !                !
         !        Name Strings           !                !
         !                               !                !
         !                               !                !
         !                               !                !
         +-------------------------------+                !
         !                               !                !
         +--                           --+                !
         !       CRC Check Value         !                !
         +--                           --+                !
         !                               !                !
         +-------------------------------+   -------------+---
```

## SINGLE DENSITY FLOPPY DISK FORMAT

| | | | | |
|---|---|---|---|---|
| SIZE | 5" | | 8" | |
| DENSITY | SINGLE | | SINGLE | |
| #TRACKS | 35 | | 77 | |
| #SECTORS/TRACK | 10 | | 16 | |
| BYTES/TRACK (UNFORMATTED) | 3125 | | 5208 | |
| FORMAT | #BYTES (DEC) | VALUE (HEX) | #BYTES (DEC) | VALUE (HEX) |
| HEADER (ONCE PER TRACK) | 30 | FF | 30 | FF |
| | 6 | 00 | 6 | 00 |
| | 1 | FC | 1 | FC |
| | 12 | FF | 12 | FF |
| SECTOR (REPEATED N TIMES) | 6 | 00 | 6 | 00 |
| | 1 | FE | 1 | FE |
| | 1 | (TRK #) | 1 | (TRK #) |
| | 1 | (SIDE #) | 1 | (SIDE #) |
| | 1 | (SECT #) | 1 | (SECT #) |
| | 1 | (BYTCNT) | 1 | (BYTCNT) |
| | 1 | F7 (2 CRC) | 1 | F7 (2 CRC) |
| | 10 | FF | 10 | FF |
| | 6 | 00 | 6 | 00 |
| | 1 | FB | 1 | FB |
| | 256 | (DATA) | 256 | (DATA) |
| | 1 | F7 (2 CRC) | 1 | F7 (2 CRC) |
| | 10 | FF | 10 | FF |
| TRAILER (ONCE PER TRACK) | 96 | FF | 391 | FF |
| BYTES/SECTOR (FORMATTED) | 256 | | 256 | |
| BYTES/TRACK (FORMATTED) | 2560 | | 4096 | |
| BYTES/DISK (FORMATTED) | 89,600 | | 315,392 | |

## DOUBLE DENSITY FLOPPY DISK FORMAT

| | | | | |
|---|---|---|---|---|
| SIZE | 5" | | 8" | |
| DENSITY | DOUBLE | | DOUBLE | |
| #TRACKS | 35 | | 77 | |
| #SECTORS/TRACK | 16 | | 28 | |
| BYTES/TRACK (UNFORMATTED) | 6250 | | 10,416 | |
| FORMAT | BYTES (DEC) | VALUE (HEX) | BYTES (DEC) | VALUE (HEX) |
| HEADER (ONCE PER TRACK) | 80 | 4E | 80 | 4E |
| | 12 | 00 | 12 | 00 |
| | 3 | F5 (A1) | 3 | F5 |
| | 1 | FC | 1 | FC |
| | 32 | 4E | 32 | 4E |
| SECTOR (REPEATED N TIMES) | 12 | 00 | 12 | 00 |
| | 3 | F5 | 3 | F5 |
| | 1 | FE | 1 | FE |
| | 1 | (TRK #) | 1 | (TRK #) |
| | 1 | (SIDE #) | 1 | (SIDE #) |
| | 1 | (SECT #) | 1 | (SECT #) |
| | 1 | (BYTCNT) | 1 | (BYTCNT) |
| | 1 | F7 (2 CRC) | 1 | F7 (2 CRC) |
| | 22 | 4E | 22 | 4E |
| | 12 | 00 | 12 | 00 |
| | 3 | F5 (A1) | 3 | F5 (A1) |
| | 1 | FB | 1 | FB |
| | 256 | (DATA) | 256 | (DATA) |
| | 1 | F7 (2 CRC) | 1 | F7 (2 CRC) |
| | 22 | 4E | 22 | 4E |
| TRAILER (ONCE PER TRACK) | 682 | 4E | 768 | 4E |
| BYTES/SECTOR (FORMATTED) | 256 | | 256 | |
| BYTES/TRACK (FORMATTED) | 4096 | | 7168 | |
| BYTES/DISK (FORMATTED) | 141,824 | | 548,864 | |

```
*****************************************************************
*                                                               *
*              CLOCK MODULE FOR THE MPT TIMER                   *
*              (C) 1981  Microware Systems Corporation          *
*                                                               *
*****************************************************************

         NAM    Clock Module
         TTL    Definitions
         Use    /d0/defs/systype
         opt    -c

***************************
* System Type Definitions *
***************************

CPUTYP   SET    GIMIX       CPU type
DSKTYP   SET    DCB4        Disk Controller type
CLKTYP   SET    MPT         Clock type
INTRPT   SET    YES         Interrupt Driven Disk Flag
DRVCNT   SET    4           Number of Drive Descriptors
DRVSIZ   SET    8           Drive size
REV      SET    1           Revision Level

***************************
*  Disk Port Address      *
***************************

DPORT    SET    Ø
DPORT    SET    $E6ØØ

***************************
*  Clock Port Address     *
***************************

CPORT    SET    Ø
CPORT    SET    $EØ5Ø

***************************
*  I/O Port Addresses     *
***************************

A.TERM   SET    $EØØ4       ACIA Master Terminal
A.T1     SET    $EØ2Ø       ACIA Secondary Terminal
A.P      SET    $EØ4Ø       PIA Printer (B-side)
PIASID   set    a.side
A.P1     SET    $EØ3Ø       ACIA Printer

         opt    c
         opt    -c
```

```
***************************
*     MODULE HEADER       *
***************************

Type     SET   SYSTM+OBJCT
Revs     SET   REENT+1
ClkMod   Mod   ClkEnd,ClkNam,Type,Revs,ClkEnt,CPORT
ClkNam   FCS   /Clock/
         FCB   2           Edition number
CLKPRT   EQU   M$STAK      Stack has Clock Port address

***************************
* CLOCK DATA DEFINITIONS *
***************************

TIMSVC   FCB   F$TIME
         FDB   TIME-*-2
         FCB   $80

***************************
*  DAYS IN MONTHS TABLE   *
***************************

MONTHS   FCB   0           UNINITIALIZED MONTH
         FCB   31          JANUARY
         FCB   28          FEBRUARY
         FCB   31          MARCH
         FCB   30          APRIL
         FCB   31          MAY
         FCB   30          JUNE
         FCB   31          JULY
         FCB   31          AUGUST
         FCB   30          SEPTEMBER
         FCB   31          OCTOBER
         FCB   30          NOVEMBER
         FCB   31          DECEMBER

*************************************
* CLOCK INTERRUPT SERVICE ROUTINE *
*************************************

NOTCLK   JMP   [D.ISVC]    GO TO INTERRUPT SERVICE

CLKSRV   LDX   CLKPRT,PCR  GET CLOCK ADDRESS
         LDA   1,X         GET CONTROL REGISTER
         BITA  #$80        IS IT CLOCK?
         BEQ   NOTCLK      BRANCH IF NOT
         LDA   0,X         CLEAR CLOCK INTERRUPT

TICK     CLRA              SET DIRECT PAGE
         TFR   A,DP
```

```
****************************
*  UPDATE CURRENT TIME     *
****************************

         DEC   D.TIC        COUNT TICK
         BNE   TICK50       BRANCH IF NOT END OF SECOND
         LDD   D.MIN        GET MINUTE & SECOND
         INCB               COUNT SECOND
         CMPB  #60          END OF MINUTE?
         BCS   TICK35       BRANCH IF NOT
         INCA               COUNT MINUTE
         CMPA  #60          END OF HOUR?
         BCS   TICK30       BRANCH IF NOT
         LDD   D.DAY        GET DAY & HOUR
         INCB               COUNT HOUR
         CMPB  #24          END OF DAY?
         BCS   TICK25       BRANCH IF NOT
         INCA               COUNT DAY
         LEAX  MONTHS,PCR   GET DAYS/MONTH TABLE
         LDB   D.MNTH       GET MONTH
         CMPB  #2           IS IT FEBRUARY?
         BNE   TICK10       BRANCH IF NOT
         LDB   D.YEAR       GET YEAR
         BEQ   TICK10       BRANCH IF EVEN HUNDRED
         ANDB  #3           IS IT LEAP YEAR?
         BNE   TICK10       BRANCH IF NOT
         DECA               ADD FEB 29
TICK10   LDB   D.MNTH       GET MONTH
         CMPA  B,X          END OF MONTH?
         BLS   TICK20       BRANCH IF NOT
         LDD   D.YEAR       GET YEAR & MONTH
         INCB               COUNT MONTH
         CMPB  #13          END OF YEAR?
         BCS   TICK15       BRANCH IF NOT
         INCA               COUNT YEAR
         LDB   #1           NEW MONTH
TICK15   STD   D.YEAR       UPDATE YEAR & MONTH
         LDA   #1           NEW DAY
TICK20   CLRB               NEW HOUR
TICK25   STD   D.DAY        UPDATE DAY & HOUR
         CLRA               NEW MINUTE
TICK30   CLRB               NEW SECOND
TICK35   STD   D.MIN        UPDATE MINUTE & SECOND
         LDA   D.TSEC       GET TICKS/SECOND
         STA   D.TIC
TICK50   JMP   [CLOCK]      GO TO SYSTEM CLOCK ROUTINE
```

```
*********************************
*  CLOCK INITIALIZATION ENTRY  *
*********************************

ClkEnt   PSHS  DP           save Direct Page
         CLRA               clear DP
         TFR   A,DP
         PSHS  CC           save interrupt masks
         LDA   #10          SET TICKS / SECOND
         STA   D.TSEC
         STA   D.TIC
         LDA   #1           SET TICKS / TIME-SLICE
         STA   D.TSLC
         STA   D.SLIC
         ORCC  #IRQM+FIRQM SET INTRPT MASKS
         LEAX  CLKSRV,PCR GET SERVICE ROUTINE
         STX   D.IRQ        SET INTERRUPT VECTOR
         LDX   CLKPRT,PCR get clock address
         CLRA
         CLRB
         STD   0,X          CLEAR PIA REGS.
         LDD   #$FF3D       INITIALIZE CLOCK BOARD
         STD   0,X
         LDD   #$8005
         STA   0,X
         STB   0,X
         LDA   0,X          CLEAR ANY INTERRUPTS
         PULS  CC           retrieve masks
         LEAY  TIMSVC,PCR
         OS9   F$SSVC       SET TIME SEVICE ROUTINE
         PULS  DP,PC

*************************
*    SUBROUTINE TIME    *
*  (RETURN TIME OF DAY) *
*************************

TIME     EQU   *
         LDX   R$X,U        Get Specified location
         LDD   D.YEAR       Get Year & Month
         STD   0,X
         LDD   D.DAY        Get Day & Hour
         STD   2,X
         LDD   D.MIN        Get Minute & Second
         STD   4,X
         CLRB               Clear Carry
         RTS

         EMod
ClkEnd   EQU   *
         END
```

```
**************************************************************
*                                                            *
*          TERM -- Device Descriptor Module                  *
*          (C) 1981  Microware Systems Corporation           *
*                                                            *
**************************************************************

        mod    TRMEND,TRMNAM,DEVIC+OBJCT,REENT+1,TRMMGR,
        fcb    UPDAT.      mode
        fcb    $FF
        fdb    A.TERM      port addresss
        fcb    TRMNAM-*-1  option byte count
        fcb    DT.SCF      Device Type: SCF

* DEFAULT PARAMETERS

        fcb    0           case=UPPER and lower
        fcb    1           backspace=BS,SP,BS
        fcb    0           delete=backspace over line
        fcb    1           auto echo on
        fcb    1           auto line feed on
        fcb    0           null count
        fcb    1           end of page pause on
        fcb    24          lines per page
        fcb    C$BSP       backspace char
        fcb    C$DEL       delete line char
        fcb    C$CR        end of record char
        fcb    C$EOF       end of file char
        fcb    C$RPRT      reprint line char
        fcb    C$RPET      dup last line char
        fcb    C$PAUS      pause char
        fcb    C$INTR      Keyboard Interrupt char
        fcb    C$QUIT      Keyboard Quit char
        fcb    C$BSP       backspace echo char
        fcb    C$BELL      line overflow char
        fcb    $15         no parity
        fcb    0           undefined baud rate
        fdb    TRMNAM      offset of echo device
TRMNAM  fcs    "TERM"      device name
TRMMGR  fcs    "SCF"        file manager
TRMDRV  fcs    "ACIA"      device driver

        emod               Module CRC

TRMEND  EQU    *
```

```
***********************************************************
*                                                         *
*       ACIA  - Interrupt Driven ACIA Device Driver       *
*       (C) 1981  Microware Systems Corporation           *
*                                                         *
***********************************************************

         NAM    ACIA
         ifp1
         endc               d0/defs/scfdefs

INPSIZ   set    100         input  buffer SIZE (<=256)
OUTSIZ   set    40          output buffer SIZE (<=256)

PARITY   set    %01000000   parity  error bit
OVERUN   set    %00100000   overrun error bit
FRAME    set    %00010000   framing error bit
NOTCTS   set    %00001000   not clear to send
DCDLST   set    %00000100   data carrier lost

INPERR   set    PARITY+OVERUN+FRAME+NOTCTS+DCDLST

***************************
* Static storage offsets *
***************************

         ORG    V.SCF       room for SCF variables
INXTI    RMB    1           input  buffer NEXT-IN  ptr
INXTO    RMB    1           input  buffer NEXT-OUT ptr
ONXTI    RMB    1           output buffer NEXT-IN  ptr
ONXTO    RMB    1           output buffer NEXT-OUT ptr
INPBUF   RMB    INPSIZ      input  buffer
OUTBUF   RMB    OUTSIZ      output buffer
ACIMEM   EQU    .           TOTAL STATIC STORAGE REQUIREME

****************************
*    MODULE HEADER         *
****************************

         MOD    ACIEND,ACINAM,DRIVR+OBJCT,REENT+1,ACIENT,
         FCB    UPDAT.
ACINAM   FCS    "ACIA"

         fcb    2           editiion number

****************************
*     BRANCH TABLE         *
****************************

ACIENT   LBRA   INIT
         LBRA   READ
         LBRA   WRITE
         LBRA   GETSTA
         LBRA   PUTSTA
         LBRA   TRMNAT
```

```
ACMASK   FCB   0            no FLIP bits
         FCB   $80          IRQ POLLING MASK
         FCB   5            (low) PRIORITY

*******************************
* INITIALIZE (TERMINAL) ACIA *
*******************************

INIT     LDX   V.PORT,U     I/O port address
         LDB   #$03         master reset signal
         STB   0,X          reset ACIA
         LDA   M$OPT,Y      get option byte count
         CMPA  #PD.PAR-PD.OPT acia control value given?
         BLO   INIT10       ..No; default $15
         LDB   PD.PAR-PD.OPT+M$DTYP,Y
         BNE   INIT20
INIT10   LDB   #$15         default acia control
INIT20   STB   V.TYPE,U     save device type
         LDD   V.PORT,U
         LEAX  ACMASK,PCR
         LEAY  ACIRQ,PCR    address of INTERRUPT SERVICE R
         OS9   F$IRQ        ADD to IRQ POLLING TABLE
         BCS   INIT9        ERROR - return it
         CLRA
         CLRB
         STD   INXTI,U      INITIALIZE buffer ptrs
         STD   ONXTI,U
         LDX   V.PORT,U
INIT30   LDB   V.TYPE,U
         ORB   #$80         Enable ACIA input interrupts
         STB   0,X          initialize ACIA for input inte
INIT9    RTS                return (carry clear)

**************************************************
* READ:  return ONE BYTE of input from the ACIA *
*                                               *
* PASSED: (Y)=PATH DESCRIPTOR                   *
*         (U)=STATIC STORAGE address            *
* RETURNS: (A)=input BYTE (carry clear)         *
*      or   CC=SET, B=ERROR code if error       *
**************************************************

READ00   BSR   ACSLEP       wait for acia data
READ     LDB   INXTO,U      (input buffer) NEXT-OUT ptr
         LEAX  INPBUF,U     address of input buffer
         ORCC  #IRQM        calm interrupts
         CMPB  INXTI,U      any data AVAILABLE?
         BEQ   READ00       ..No; wait, and retry
         ABX
         LDA   0,X          the char
         INCB               ADVANCE NEXT-OUT ptr
         CMPB  #INPSIZ-1    end of circular buffer?
         BLS   READ10       ..No
         CLRB               reset ptr to start of buffer
READ10   STB   INXTO,U      save updated Buffer ptr
         CLRB
```

```
         LDB    V.ERR,U     Transmission error?
         BEQ    READ90      ..No; return
         STB    PD.ERR,Y    return error bits in PD
         CLR    V.ERR,U
         COMB               return carry set
         LDB    #E$RD       signal read error
READ90   ANDCC  #$FF-IRQM   enable IRQ requests
         RTS

*********************************************
* ACSLEP - Sleep for I/O activity           *
* This version HOGS CPU if signal pending   *
*                                           *
* PASSED: (cc)=IRQ's MUST be disabled       *
*         (U)=Global Storage                *
*         V.BUSY,U=current proc id          *
* DESTROYS: possibly PC                     *
*********************************************

ACSLEP   PSHS   D,X
         LDA    V.BUSY,U    get current process id
         STA    V.WAKE,U    arrange wake up signal
         ANDCC  #$FF-IRQM   interrupts ok now
         LDX    #0
         OS9    F$SLEP      wait for input data
         LDX    D.PROC
         LDB    P$SIGN,X    signal present?
         beq    ACSL90      ..No; return
         cmpb   #S$INTR     Deadly signal?
         bls    ACSLER      ..Yes; return error
ACSL90   CLRB               clear carry
         PULS   D,X,PC      return
 ACSLER   LEAS   6,S         Exit to caller's caller
         COMA               return carry set
         RTS

**************************************************
* WRITE char THROUGH ACIA                        *
*                                                *
* PASSED: (A)=char to write                      *
*         (Y)=PATH DESCRIPTOR                    *
*         (U)=STATIC STORAGE address             *
* RETURNS: CC=SET IF BUSY (output buffer FULL)   *
**************************************************

WRIT00   BSR    ACSLEP      sleep a bit
WRITE    LEAX   OUTBUF,U    output buffer address
         LDB    ONXTI,U     (output) NEXT-OUT ptr
         ABX
         STA    0,X         PUT char in buffer
         INCB               ADVANCE the ptr
         CMPB   #OUTSIZ-1   end of circular buffer?
         BLS    WRIT10      ..No
         CLRB               reset ptr to start of buffer
WRIT10   ORCC   #IRQM       disable interrupts
         CMPB   ONXTO,U     buffer FULL?
```

```
         BEQ    WRITØØ        ..Yes; sleep and retry
         STB    ONXTI,U       save updated NEXT-IN ptr
         ANDCC  #$FF-IRQM     enable IRQs
         LDA    V.TYPE,U      PARITY CONTROL
         ORA    #$AØ          ENABLE input/output IRQS
         STA    [V.PORT,U]    ENABLE INTERRUPTS
WRIT9Ø   CLRB                 (return carry clear)
         RTS

*************************************
* GET/PUT ACIA STATUS               *
*                                   *
* PASSED: (A)=STATUS-CODE           *
*         (Y)=PATH DESCRIPTOR       *
*         (U)=STATIC STORAGE address *
* RETURNS: varies                   *
*************************************

GETSTA   CMPA   #SS.RDY       READY STATUS?
         BNE    GETS1Ø        ..No
         LDA    INXTO,U
         SUBA   INXTI,U       any data AVAILABLE?
         BNE    WRIT9Ø        ..Yes; return carry clear
         COMB
         LDB    #E$NRDY
         RTS
GETS1Ø   CMPA   #SS.EOF       End of file?
         BEQ    WRIT9Ø        ..Yes; Return carry clear

PUTSTA   COMB                 return carry set
         LDB    #E$USVC       UNKNOWN SERVICE CODE
         RTS

*****************************
* TERMINATE ACIA processing *
*                           *
* PASSED: (U)=STATIC STORAGE *
* RETURNS: NOTHING          *
*****************************

TRMNØØ   BSR    ACSLEP        wait for I/O activity
TRMNAT   LDX    D.PROC
         LDA    P$ID,X
         STA    V.BUSY,U
         STA    V.LPRC,U
         LDB    ONXTI,U
         ORCC   #IRQM         disable interrupts
         CMPB   ONXTO,U       output done?
         BEQ    TRMNØØ        ..No; sleep a bit
         LDA    #$Ø3
         STA    [V.PORT,U]    disable ACIA interrupts
         ANDCC  #$FF-IRQM     enable interrupts
         LDX    #Ø
         OS9    F$IRQ         remove acia from polling tbl
         RTS
```

```
****************************************************************
* ACIRQ:  Process INTERRUPT (input or output) from ACIA        *
*                                                              *
* PASSED: (U)=STATIC STORAGE addr                              *
*         (X)=Port address                                     *
*         (A)=polled status                                    *
* Returns: NOTHING                                             *
****************************************************************

ACIRQ    LDX   V.PORT,U    get port address
         ANDA  #INPERR     mask status error bits
         ORA   V.ERR,U
         STA   V.ERR,U     update cumulative errors
         LDA   0,X         restore acia status
         BITA  #1          input ready?
         BNE   INACIA      ..yes; go get it

*********************************************
* FALL THROUGH to DO output                 *
*                                           *
* OACIA:   Output to ACIA INTERRUPT ROUTINE *
*                                           *
* PASSED: (A)=ACIA STATUS REGISTER CONTENTS *
*         (X)=ACIA port address             *
*         (U)=STATIC STORAGE address        *
*********************************************

OACIA    LEAY  OUTBUF,U    output buffer ptr
         LDB   ONXTO,U     (output) NEXT-OUT ptr
         cmpb  ONXTI,U     output buffer already empty?
         beq   OACIA2      ..Yes; disable output IRQ, ret
         CLRA
         LDA   D,Y         next output char
         INCB              ADVANCE NEXT-OUT ptr
         CMPB  #OUTSIZ-1   end of circular buffer?
         BLS   OACIA1      ..No
         CLRB
OACIA1   STB   ONXTO,U     save updated NEXT-OUT ptr
         STA   1,X         WRITE the char
         CMPB  ONXTI,U     last char in output buffer?
         BNE   WAKEUP      ..No
OACIA2   LBSR  INIT30      disable output rdy IRQ

WAKEUP   LDB   #S$WAKE     WAKE UP SIGNAL
         LDA   V.WAKE,U    OWNER WAITING?
WAKE10   BEQ   WAKE90      ..No; return
         OS9   F$SEND
WAKE90   clr   V.WAKE,U
         RTS
```

```
*********************************************
* INACIA:    Process ACIA input INTERRUPT    *
*                                            *
* PASSED: (A)=ACIA STATUS REGISTER data      *
*         (X)=ACIA port address              *
*         (U)=STATIC STORAGE address         *
*                                            *
* NOTICE the ABSENCE of ERROR CHECKING HERE *
*********************************************

INACIA   LDA    1,X          READ input char
         LEAX   INPBUF,U     input buffer
         LDB    INXTI,U      (input) NEXT-IN ptr
         ABX
         STA    0,X          save char in buffer
         INCB                update NEXT-IN ptr
         CMPB   #INPSIZ-1    end of circular buffer?
         BLS    ACIA2        ..No
         CLRB
ACIA2    CMPB   INXTO,U      input OVERRUN?
         BNE    ACIA25       ..No; good
         LDB    #OVERUN      mark overrun error
         ORB    V.ERR,U
         STB    V.ERR,U
         BRA    ACIA26       throw away character
ACIA25   STB    INXTI,U      update NEXT-IN ptr
ACIA26   ANDA   #$7F
         BEQ    WAKEUP       ..pass nulls without ctl check
         CMPA   V.PCHR,U     PAUSE char?
         BNE    ACIA3        ..No
         LDX    V.DEV2,U     PAUSE DEVICE STATIC
         BEQ    WAKEUP       ..None
         STA    V.PAUS,X     REQUEST PAUSE
         BRA    WAKEUP

ACIA3    LDB    #S$INTR      INTERRUPT SIGNAL
         CMPA   V.INTR,U     keyboard INTERRUPT SIGNAL?
         BEQ    ACIA4        ..Yes
         LDB    #S$ABT       ABORT SIGNAL
         CMPA   V.QUIT,U     keyboard ABORT SIGNAL?
         BNE    WAKEUP       ..No
ACIA4    LDA    V.LPRC,U     last process ID
         BRA    WAKE10       SEND ERROR SIGNAL

         emod                Module CRC

ACIEND   EQU    *
```

```
*****************************************************************
*                                                               *
*         PRINTERR - English Error Printer Module               *
*         (C) 1981  Microware Systems Corporation               *
*                                                               *
*****************************************************************

          nam    Printerr

          use    /d0/defs/os9defs
          ifp1
          endc

***********************************************************
* Printerr                                                *
*   Translate OS-9 error numbers to English messages      *
* Author: Bob Doggett                                     *
*                                                         *
* Replaces OS-9 PRTERR service routine.                   *
* Note: once this is done, there is provision             *
* for returning to OS-9's original error routine.         *
*                                                         *
* Speed could be improved, using fixed-length             *
* random file.  The text file format used by this         *
* version may be edited, and is shorter than              *
* a random file would be.                                 *
*                                                         *
* CAUTION: this version uses quite a chunk of             *
*  User's stack, and may be unsuitable for some           *
*  processes.                                             *
***********************************************************

          mod    PEREND,PERNAM,PRGRM+OBJCT,REENT+1,PRTERR,
PERNAM    fcs    "Printerr"

          fcb    3             edition number
BUFSIZ    SET    80            ERRMSG FILE MAX RCD LENGTH
C$CR      SET    $0D

*****************************************
* Execution-Time Stack temporary storage *
*****************************************

          ORG    0
PRTPTH    RMB    1             User's Std Error path
ERRPTH    RMB    1             ERRMSG path number
ERRNUM    RMB    1             Error number
BUFPTR    RMB    2             Line buffer ptr
IOBUFF    RMB    BUFSIZ        Line buffer
PERMEM    EQU    .
ERRFIL    FCC    "/D0/"
          FCC    "ERRMSG"
          FCB    C$CR

ERRMSG    FCC    "Error #"
          FCB    -1
```

```
SVCTBL    equ    *             Replacement for SYS call vecto
          fcb    F$PERR
          fdb    PERROR-*-2
          fcb    $80           end of table


**************************************************************
* Printerr:                                                  *
*    Translate OS-9 errors to English messages               *
*    using message strings in ERRMSG file                    *
*                                                            *
* Format of ERRMSG file:                                     *
*   Number   (0-3)   Ascii error number (0-255)              *
*   Delim       1    Any byte <= $2F                         *
*   Message  (0-n)   Variable length message string          *
**************************************************************

PRTERR    CLRA
          LEAX   <PERNAM,PCR
          OS9    F$LINK        Link to self
          BCS    EXIT          ..Error; Fail
          LEAY   <SVCTBL,PCR
          OS9    F$SSVC        redirect system prterr routine
          CLRB
EXIT      OS9    F$EXIT
PERROR    LDX    D.PROC
          LDA    P$PATH+2,X Get user´s std error path
          BEQ    PERR90        ..None; exit
          LEAS   -PERMEM,S     chop out temp storage
          LDB    R$B,U
          LEAU   0,S
          STA    PRTPTH,U
          STB    ERRNUM,U      save error number
          BSR    PRTNUM        print "Error #n"
          LDA    #READ.
          LEAX   ERRFIL,PCR ERRMSG file name
          OS9    I$OPEN
          STA    ERRPTH,U      save path number
          BCS    PERR90        ..ERROR; exit
          BSR    SEARCH        find error in ERRMSG file
          BNE    PERR80        ..Not found; exit
          BSR    PRTLIN        print Error Message

PERR80    LDA    ERRPTH,U
          OS9    I$CLOS        close ERRMSG file
PERR90    LEAS   PERMEM,S
          RTS

SEARCH    LDA    ERRPTH,U
          LEAX   IOBUFF,U
          LDY    #BUFSIZ
          OS9    I$RDLN        read one ERRMSG RCD
          BCS    SEAR90        ..ERROR; EXIT
          BSR    GETNUM        get number in I/O buffer
          CMPA   #'0           Followed by separator?
```

```
         BHS    SEARCH       ..no; skip this record
         CMPB   ERRNUM,U     Is this the Error number?
         BNE    SEARCH       ..No; REPEAT
SEAR90   RTS                 RETURN

*********************************************
* PRTNUM:  PRINT 8-BIT ASCII NUMBER IN (,X+) *
*********************************************

PRTNUM   LEAX   ERRMSG,PCR ptr to "Error #"
         LEAY   IOBUFF,U
         LDA    ,X+
PRTN05   STA    ,Y+
         LDA    ,X+
         BPL    PRTN05
         LDB    ERRNUM,U
         LDA    #'0-1
PRTN10   INCA                form hundreds digit
         SUBB   #100
         BCC    PRTN10
         STA    ,Y+          Put hundreds digit in buffer
         LDA    #'9+1
PRTN20   DECA                form tens digit
         ADDB   #10
         BCC    PRTN20
         STA    ,Y+          Put tens digit in buffer
         TFR    B,A
         ADDA   #'0          form units digit
         LDB    #C$CR
         STD    ,Y+          Put units digit in buffer
         LEAX   IOBUFF,U
PRTLIN   LDY    #80
PRINT    LDA    PRTPTH,U     Print to STD Error path
         OS9    I$WRLN
         RTS

GETNUM   CLRB
GETN10   LDA    ,X+
         SUBA   #'0
         CMPA   #9           Numeric character?
         BHI    GETN90       ..No; done
         PSHS   A            save digit
         LDA    #10
         MUL                 multiply partial result by 10
         ADDB   ,S+          add in next digit
         BCC    GETN10       ..Continue until overflow
GETN90   LDA    -1,X         retreive separator character
         RTS

         emod                Module CRC

PEREND   EQU    *
```

# SERVICE REQUEST SUMMARY

## USER MODE FUNCTION REQUESTS

| CODE | MNEMONIC | FUNCTION | PAGE |
|---|---|---|---|
| 103F ØØ | F$LINK | Link to memory module | 63 |
| 103F Ø1 | F$LOAD | Load module from mass-storage | 64 |
| 103F Ø2 | F$UNLK | Unlink module | 77 |
| 103F Ø3 | F$FORK | Start new process | 59 |
| 103F Ø4 | F$WAIT | Wait for signal | 78 |
| 103F Ø5 | F$CHAN | Chain process to new module | 53 |
| 103F Ø6 | F$EXIT | Terminate Process | 58 |
| 103F Ø7 | F$MEM | Set memory size | 65 |
| 103F Ø8 | F$SEND | Send signal to process | 69 |
| 103F Ø9 | F$ICPT | Set signal intercept trap | 61 |
| 103F ØA | F$SLEP | Suspend process | 7Ø |
| 103F ØB | | Not implemented | |
| 103F ØC | F$ID | Return process ID | 62 |
| 103F ØD | F$SPRI | Set process priority | 71 |
| 103F ØE | F$SSWI | Set software interrupt vector | 74 |
| 103F ØF | F$PERR | Print error message | 66 |
| 103F 1Ø | F$PNAM | Parse pathlist name | 67 |
| 103F 11 | F$CNAM | Compare two names | 55 |
| 103F 12 | F$SBIT | Search a bit map | 68 |
| 103F 13 | F$ABIT | Allocate in a bit map | 52 |
| 103F 14 | F$DBIT | Deallocate in a bit map | 57 |
| 103F 15 | F$TIME | Return current time | 76 |
| 103F 16 | F$STIM | Set current time | 75 |
| 103F 17 | F$CRC | Generate CRC | 56 |

## SYSTEM MODE PRIVILEGED FUNCTION REQUESTS

| CODE | MNEMONIC | FUNCTION | PAGE |
|---|---|---|---|
| 103F 28 | F$SRQM | System memory request | 87 |
| 103F 29 | F$SRTM | System memory return | 88 |
| 103F 2A | F$IRQ | Enter IRQ polling table | 84 |
| 103F 2B | F$IOQU | Enter I/O queue | 83 |
| 103F 2C | F$APRC | Enter active process queue | 8Ø |
| 103F 2D | F$NPRC | Start next process | 85 |
| 103F 2E | F$VMOD | Validate module | 89 |
| 103F 2F | F$F64 | Find 64 byte memory block | 81 |
| 103F 3Ø | F$A64 | Allocate a 64 byte memory block | 79 |
| 103F 31 | F$R64 | Return a 64 byte memory block | 86 |
| 103F 32 | F$SSVC* | Install a function request | 72 |
| 103F 33 | F$IODL | Delete I/O module | 82 |

*NOTE: F$SSVC is a user mode function, although its code > $27

## INPUT/OUTPUT SERVICE REQUESTS

| CODE | MNEMONIC | FUNCTION | PAGE |
|---|---|---|---|
| 103F 80 | I$ATCH | Attach I/O device | 90 |
| 103F 81 | I$DTCH | Detach I/O device | 96 |
| 103F 82 | I$DUP | Duplicate path | 97 |
| 103F 83 | I$CREA | Create a new file | 93 |
| 103F 84 | I$OPEN | Open a path to an existing file | 103 |
| 103F 85 | I$MDIR | Make a directory file | 102 |
| 103F 86 | I$CDIR | Change working directory | 91 |
| 103F 87 | I$DLET | Delete a file | 95 |
| 103F 88 | I$SEEK | Reposition file pointer | 106 |
| 103F 89 | I$READ | Read date | 104 |
| 103F 8A | I$WRIT | Write data | 110 |
| 103F 8B | I$RDLN | Read line | 105 |
| 103F 8C | I$WRLN | Write line | 111 |
| 103F 8D | I$GSTT | Get device status | 98 |
| 103F 8E | I$SSTT | Set device status | 107 |
| 103F 8F | I$CLOS | Close a path | 92 |

## STANDARD I/O PATHS

0 = Standard Input
1 = Standard Output
2 = Standard Error Output

## FILE ACCESS CODES

READ = $01
WRITE = $02
UPDATE = READ + WRITE
EXEC = $04
PREAD = $08
PWRIT = $10
PEXEC = $20
SHARE = $40
DIR = $80

## MODULE TYPES

$1 = Program
$2 = Subroutine Module
$3 = Multi-Module
$4 = Data
$C = System Module
$D = File Manager
$E = Device Driver
$F = Device Descriptor

## MODULE LANGUAGES

$0 = Data
$1 = 6809 Object code
$2 = BASIC09 I-Code
$3 = Pascal P-Code

## MODULE ATTRIBUTES

$8 = Reentrant

# OS-9 ERROR CODES

The error codes are shown in both hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

| HEX | DEC | |
|---|---|---|
| $C8 | 200 | PATH TABLE FULL - The file cannot be opened because the system path table is currently full. |
| $C9 | 201 | ILLEGAL PATH NUMBER - Number too large or for non-existant path. |
| $CA | 202 | INTERRUPT POLLING TABLE FULL |
| $CB | 203 | ILLEGAL DEVICE - Can't find device descriptor, file manager or device driver. |
| $CC | 204 | DEVICE TABLE FULL - Can't add another device. |
| $CD | 205 | ILLEGAL MODULE HEADER - Module's sync code, check character or CRC is incorrect. |
| $CE | 206 | MODULE DIRECTORY FULL - Can't add another module |
| $CF | 207 | MEMORY FULL - Not enough contiguous RAM available to process request. |
| $D0 | 208 | ILLEGAL SERVICE REQUEST - System call had an illegal code number. |
| $D1 | 209 | MODULE BUSY - non-sharable module is in use by another process. |
| $D2 | 210 | BOUNDARY ERROR - Memory allocation or deallocation request not on page boundary. |
| $D3 | 211 | END OF FILE - End of file encountered on read. |
| $D4 | 212 | NOT YOUR MEMORY - attempted to deallocate memory not previously assigned. |
| $D5 | 213 | NON-EXISTING SEGMENT - device has damaged file structure. |
| $D6 | 214 | NO PERMISSION - you don't have owner's permission to access the file as requested. |
| $D7 | 215 | BAD PATH NAME - syntax error in pathlist. |

OS-9 ERROR CODES (continued)

| HEX | DEC | |
|---|---|---|
| $D8 | 216 | MISSING PATHLIST - expected pathlist missing or in error. |
| $D9 | 217 | SEGMENT LIST FULL - file is too fragmented to be expanded further. |
| $DA | 218 | FILE ALREADY EXISTS - file name already appears in current directory. |
| $DB | 219 | ILLEGAL BLOCK ADDRESS - device's file structure has been damaged. |
| $DC | 220 | ILLEGAL BLOCK SIZE - device's file structure has been damaged. |
| $DD | 221 | MODULE NOT FOUND - request for link to module not found in directory. |
| $DE | 222 | SECTOR OUT OF RANGE - device file structure damaged or incorrectly formatted. |
| $DF | 223 | SUICIDE ATTEMPT - request to return memory where your stack is located. |
| $E0 | 224 | ILLEGAL PROCESS NUMBER - no such process exists. |
| $E1 | 225 | ILLEGAL SIGNAL CODE. |
| $E2 | 226 | NO CHILDREN - can't wait because process has no children. |
| $E3 | 227 | ILLEGAL SWI CODE - must be 1 to 3. |
| $E4 | 228 | KEYBOARD ABORT - process aborted by signal code 2. |
| $E5 | 229 | PROCESS TABLE FULL - can't fork now. |
| $E6 | 230 | ILLEGAL PARAMETER AREA - high and low bounds passed in fork call are incorrect. |
| $E7 | 231 | BACKTRACK ERROR - you'll never see this one. |

OS-9 ERROR CODES (continued)

| HEX | DEC | |
|---|---|---|
| $E8 | 234 | SIGNAL ERROR - receiving process has previous unprocessed signal pending. |

-- DEVICE DRIVER/HARDWARE ERRORS --

| HEX | DEC | |
|---|---|---|
| $F0 | 240 | UNIT ERROR - device unit does not exist. |
| $F1 | 241 | SECTOR ERROR - sector number is out of range. |
| $F2 | 242 | WRITE PROTECT - device is write protected. |
| $F3 | 243 | CRC ERROR - CRC error on read or write verify. |
| $F4 | 244 | READ ERROR - Data transfer error during disk read operation. |
| $F5 | 245 | WRITE ERROR - hardware error during disk write operation. |
| $F6 | 246 | NOT READY - device has "not ready" status. |
| $F7 | 247 | SEEK ERROR - physical seek to non-existant sector. |
| $F8 | 248 | MEDIA FULL - insufficient free space on media. |
| $F9 | 249 | WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive) |